conference

proceedings

10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)

Lombard, IL, USA
April 2–5, 2013

Sponsored by

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

in cooperation with
ACM SIGCOMM and
ACM SIGOPS

Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation *Lombard, IL, USA April 2–5, 2013*

usenix ASSOCIATION

ISBN 978-1-931971-00-3

USENIX Association

Proceedings of NSDI '13:
10th USENIX Symposium on Networked Systems Design and Implementation

April 2–5, 2013
Lombard, IL

Symposium Organizers

Program Co-Chairs

Nick Feamster, Georgia Tech

Jeff Mogul, HP Labs

Program Committee

Aditya Akella, *University of Wisconsin—Madison*

Katerina Argyraki, *École Polytechnique Fédérale de Lausanne (EPFL)*

Suman Banerjee, *University of Wisconsin—Madison*

Matthew Caesar, *University of Illinois at Urbana-Champaign*

Jeff Chase, *Duke University*

Rodrigo Fonseca, *Brown University*

Bryan Ford, *Yale University*

Michael J. Freedman, *Princeton University*

Krishna Gummadi, *Max Planck Institute for Software Systems (MPI-SWS)*

Rebecca Isaacs, *Microsoft Research*

Jaeyeon Jung, *Microsoft Research*

Brad Karp, *University College London*

Ethan Katz-Bassett, *University of Southern California*

Dejan Kostić, *Institute IMDEA Networks*

Arvind Krishnamurthy, *University of Washington*

Wenke Lee, *Georgia Institute of Technology*

Dave Levin, *University of Maryland*

Philip Levis, *Stanford University*

Ratul Mahajan, *Microsoft Research*

James Mickens, *Microsoft Research*

Andrew Moore, *University of Cambridge*

Richard Mortier, *University of Nottingham*

Michael Piatek, *Google*

George Porter, *University of California, San Diego*

Luigi Rizzo, *Università di Pisa*

Lakshmi Subramanian, *New York University*

Renata Teixeira, *CNRS and UPMC Sorbonne Universités*

Kobus Van der Merwe, *University of Utah*

Michael Walfish, *University of Texas, Austin*

Heather Zheng, *University of California, Santa Barbara*

Poster/Demo Program Chair

Matthew Caesar, *University of Illinois at Urbana-Champaign*

Poster/Demo Program Committee

Theophilus Benson, *Duke University*

Augustin Chaintreau, *Columbia University*

Jon Crowcroft, *University of Cambridge*

Bruce Davie, *Nicira*

Nate Foster, *Cornell University*

Phillipa Gill, *Stony Brook University*

Sharon Goldberg, *Boston University*

Shyamnath Gollakota, *University of Washington*

Jaeyeon Jung, *Microsoft Research*

Charles Killian, *Purdue University*

Prateek Mittal, *University of California, Berkeley*

Sylvia Ratnasamy, *University of California, Berkeley*

Antony Rowstron, *Microsoft Research*

Vyas Sekar, *Stony Brook University*

Junfeng Yang, *Columbia University*

Steering Committee

Casey Henderson, *USENIX Association*

Brian Noble, *University of Michigan*

Jennifer Rexford, *Princeton University*

Mike Schroeder, *Microsoft Research*

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

Chandu Thekkath, *Microsoft Research*

Amin Vahdat, *University of California, San Diego*

External Reviewers

Lorenzo Alvisi
Allen Clement
Hal Daume III
Luca Deri
Roger Dingledine
Timothy Harris
Lon Ingram
David Irwin
Kyle Jamieson
Sam King
Wyatt Lloyd
Michael Mitzenmacher
Peter Peresini
Ariel Rabkin
Siddhartha Sen
Emin Gün Sirer
Chandu Tekkath
Nedeljko Vasic
John Wilkes
Keith Winstein
David Wolinsky,

NSDI '13:
10th USENIX Symposium on Networked Systems Design and Implementation
April 2–5, 2013
Lombard, IL

Wednesday, April 3, 2013

Software Defined Networking

Pervasive Computing

Network Integrity

(Wednesday, April 3, continues on p. iv)

Data Centers

Thursday, April 4, 2013

Substrate

Wireless

Performance

Big Data

Friday, April 5, 2013

Reliability

Applications

Security and Privacy

Message from the Program Co-Chairs

As Program Co-Chairs, we are pleased to welcome you to NSDI '13. This year's conference presents the best work in the area of networked systems. Our technical program contains papers spanning leading-edge topics ranging from wireless networking to big data and data center networking.

We received 170 paper submissions, close to the high-water mark of 175 submissions in 2010. Following the lead of some recent conferences, we allowed an unlimited number of pages of references, in addition to 12 pages for all other content. All submissions that met the formatting and basic quality standards were reviewed by the Program Committee, and in a small number of cases we used external reviewers to complement the expertise of the PC. Our review process included two full rounds of reviews, extensive online discussions after each round, and face-to-face discussion at the Program Committee meeting itself. During the first round of reviewing, all papers received three reviews. We then selected 84 papers for a second round of three reviews, of which 9 received additional reviews in a third round. Overall, we gathered 775 reviews[a], with an average load of 25.8 reviews per PC member. (An additional 6 papers, for which both co-chairs had conflicts, were reviewed outside our system, and we lack statistics for these papers.)

We selected 64 papers for consideration at the PC meeting. 28 of the 30 PC members attended the meeting at HP Labs on December 3, 2012, and the other two members, who were unable to travel, attended by telephone. At the meeting, the PC accepted 38 papers, including one for which both chairs were conflicted. (Neither chair was an author of an accepted paper.) We strongly encouraged the PC members to accept a larger number of papers than in the past, based on our wish not to exclude any good papers merely for reasons of space, and on our belief that the review process can skew towards excessive negativity. We also encouraged the PC to prefer the risk of mistakenly accepting papers rather than mistakenly rejecting them, in a few cases where the PC members could not agree.

Because of the special role that conferences play in our field, all accepted papers were shepherded by a Program Committee member. 13 papers were granted extra pages to address reviewers' comments, based on the approval of their shepherds. We also asked the shepherds to write short "public summaries" of the accepted papers, in order to provide some context for readers, and to explain the PC's view of the significance of the papers.

We are grateful to everyone whose hard work made this conference possible. Most of all, we are indebted to all of the authors who submitted their work to this conference. We thank the Program Committee for their dedication, timeliness, and hard work in reviewing papers and participating in the extensive discussions at the PC meeting, as well as for their efforts in the shepherding process. We also thank our external reviewers for lending their expertise on short notice. We thank Matt Caesar for acting as Poster/Demo Program Chair, and his Poster/Demo Program Committee members for their work. We also thank Richard Mortier for chairing the subcommittee to choose award-winning papers. We extend special thanks to Jessica Cheung and Alesha Cater for their help in organizing the logistics of the PC meeting at HP Labs, and HP Labs for funding the meals during the meeting.

We are grateful to the conference sponsors for their support and to the USENIX staff for handling the conference logistics, marketing, and proceedings publication; as always, it is a pleasure to work with them. We relied heavily on the HotCRP reviewing software, and we thank Eddie Kohler for his continuing willingness to support and embellish it. We thank Geoff Voelker for the Banal format checker, which reduced our load in enforcing format compliance. Finally, we thank the NSDI '13 attendees and future readers of these papers: in the end, it is your participation in our field and interest in the work that makes NSDI, and our community, a success.

Nick Feamster, ***Georgia Tech***
Jeff Mogul, ***HP Labs***
NSDI '13 Program Co-Chairs

Composing Software-Defined Networks

Christopher Monsanto[*], *Joshua Reich*[*], *Nate Foster*[†], *Jennifer Rexford*[*], *David Walker*[*]
[*]Princeton [†]Cornell

Abstract

Managing a network requires support for multiple concurrent tasks, from routing and traffic monitoring, to access control and server load balancing. Software-Defined Networking (SDN) allows applications to realize these tasks directly, by installing packet-processing rules on switches. However, today's SDN platforms provide limited support for creating modular applications. This paper introduces new abstractions for building applications out of multiple, independent modules that jointly manage network traffic. First, we define composition operators and a library of policies for forwarding and querying traffic. Our parallel composition operator allows multiple policies to operate on the same set of packets, while a novel *sequential composition operator* allows one policy to process packets after another. Second, we enable each policy to operate on an *abstract topology* that implicitly constrains what the module can see and do. Finally, we define a new *abstract packet model* that allows programmers to extend packets with virtual fields that may be used to associate packets with high-level meta-data. We realize these abstractions in Pyretic, an imperative, domain-specific language embedded in Python.

1 Introduction

Software-Defined Networking (SDN) can greatly simplify network management by offering programmers network-wide visibility and direct control over the underlying switches from a logically-centralized controller. However, existing controller platforms [7, 12, 19, 2, 3, 24, 21] offer a "northbound" API that forces programmers to reason manually, in unstructured and ad hoc ways, about low-level dependencies between different parts of their code. An application that performs multiple tasks (e.g., routing, monitoring, access control, and server load balancing) must ensure that packet-processing rules installed to perform one task do not override the functionality of another. This results in monolithic applications where the logic for different tasks is inexorably intertwined, making the software difficult to write, test, debug, and reuse.

Modularity is the key to managing complexity in any software system, and SDNs are no exception. Previous research has tackled an important special case, where each application controls its own *slice*—a *disjoint* portion of traffic, over which the tenant or application module has (the illusion of) complete visibility and control [21, 8]. In addition to traffic isolation, such a platform may also support subdivision of network resources (e.g., link bandwidth, rule-table space, and controller CPU and memory) to prevent one module from affecting the performance of another. However, previous work does not address how to build a *single* application out of multiple, independent, reusable network policies that affect the processing of the *same* traffic.

Composition operators. Many applications require the same traffic to be processed in multiple ways. For instance, an application may route traffic based on the destination IP address, while monitoring the traffic by source address. Or, the application may apply an access-control policy to drop unwanted traffic, before routing the remaining traffic by destination address. Ideally, the programmer would construct a sophisticated application out of multiple modules that each *partially* specify the handling of the traffic. Conceptually, modules that need to process the same traffic could run in parallel or in series. In our previous work on Frenetic [6, 14], we introduced *parallel* composition, which gives each module (e.g., routing and monitoring) the illusion of operating on its own copy of each packet. This paper introduces a new kind of composition—*sequential* composition—that allows one module to act on the packets already processed by another module (e.g., routing after access control).

Topology abstraction. Programmers also need ways to limit each module's sphere of influence. Rather than have a programming platform with one (implicit) global network, we introduce *network objects*, which allow

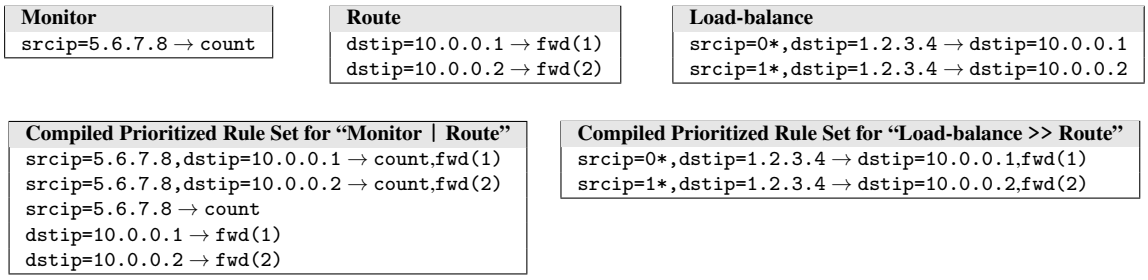


Figure 1: Parallel and Sequential Composition.

each module to operate on its own abstract view of the network. Programmers can define network objects that naturally constrain what a module can see (information hiding) and do (protection), extending previous work on topology abstraction techniques [4, 17, 10, 25].

The Pyretic Language and System. Pyretic is a new language and system that enables programmers to specify network policies at a high level of abstraction, compose them together in a variety of ways, and execute them on abstract network topologies. Running Pyretic programs efficiently relies on having a run-time system that performs composition and topology mapping to generate rules to install in the switches. Our initial prototype, built on top of POX [19], is a simple interpreter that handles each packet at the controller. While sufficient to execute and test Pyretic programs, it does not provide realistic performance. In our ongoing work, we are extending our run-time system to proactively generate and install OpenFlow rules, building on our previous research [14].

The next section presents a top-down overview of Pyretic's composition operators and topology abstraction mechanisms. The following two sections then explain each in detail, building a complete picture of Pyretic from the bottom up. Section 3 presents the Pyretic language, including an abstract packet model that conveys information between modules, and a library for defining and composing policies. Section 4 introduces network objects, which allow each module to apply a policy over its own abstract topology, and describes how our run-time system executes Pyretic programs. To evaluate the language, Section 5 presents example applications running on our Pyretic prototype. After reviewing related work in Section 6, we conclude in Section 7.

2 Abstractions for Modular Programming

Building modular SDN applications requires support for composition of multiple independent modules that each partially specify how traffic should be handled. The parallel and sequential composition operators (Section 2.1) offer simple, yet powerful, ways to combine policies generated by different modules. Network objects (Section 2.2) allow policies to operate on abstract locations that map—through one or more levels of indirection—to ones in the physical network.

2.1 Parallel and Sequential Composition Operators

Parallel and sequential composition are two central mechanisms for specifying the relationship between packet-processing policies. Figure 1 illustrates these abstractions through two examples in which policies are specified via prioritized lists of OpenFlow-like rules. Each rule includes a *pattern* (`field=value`) that matches on bits in the packet header (e.g., source and destination MAC addresses, IP addresses, and TCP/UDP port numbers), and simple *actions* the switch should perform (e.g., drop, flood, forward out a port, rewrite a header field, or count[1] matching packets). When a packet arrives, the switch (call it s) identifies the first matching rule and performs the associated actions. Note that one may easily think of such a list of rules as a function: The function input is a packet at a particular inport on s and the function output is a multiset of zero or more packets on various outports of s (zero output packets if the matching rule drops the input packet; one output if it forwards the input; and one or more if it floods). We call a packet together with its location a *located packet*.

Parallel Composition (`|`): Parallel composition gives the illusion of multiple policies operating concurrently on separate copies of the same packets [6, 14]. Given two policy functions f and g operating on a located packet p, parallel composition computes the multiset *union* of $f(p)$ and $g(p)$—that is, every located packet produced by either policy. For example, suppose a programmer writes one module to monitor traffic by source IP address, and another to route traffic by destination IP address. The monitoring module (Figure 1, top-left) comprises a simple policy that consists of a single rule applying the count action to packets matching source IP address `5.6.7.8`. The routing module (Figure 1, top-middle) consists of two rules, each matching on a destination IP address and forwarding packets out the specified port. Each module

[1] The OpenFlow API does not have an explicit `count` action; instead, every rule includes byte and packet counters. We consider `count` as an explicit action for ease of exposition.

generates its policy independently, with the programmer using the "|" operator to specify that both the route and monitor functions should be performed simultaneously. These can be mechanically compiled into a single joint ruleset (Figure 1, bottom-left) [14].

Sequential Composition (>>): Sequential composition gives the illusion of one module operating on the packets produced by another. Given two policy functions f and g operating on a located packet p, sequential composition applies g to each of the located packets produced by $f(p)$, to produce a new set of located packets. For example, suppose a programmer writes one module to load balance traffic destined to a public IP address `1.2.3.4`, over multiple server replicas at private addresses `10.0.0.1` and `10.0.0.2`, respectively, and another to route traffic based on the chosen destination server. The load-balancing module splits traffic destined to the public IP between the replicas based on the client IP address. Traffic sent by clients with an IP address whose highest-order bit is `0` go to the first server, while remaining traffic goes to the second server. As shown in Figure 1 (top-right), the load balancer performs a rewriting action to modify the destination address to correspond to the chosen server replica, without actually changing the packet's location. This load balancer can be composed sequentially with the routing policy introduced earlier. Here the programmer uses the ">>" operator to specify that load balancing should be performed first, followed by routing. Again, these may be mechanically compiled into a single joint ruleset (Figure 1, bottom-right).

2.2 Topology Abstraction With Network Objects

Modular programming requires a way to constrain what each module can *see* (information hiding) and *do* (protection). Network objects offer both information hiding and protection, while offering the familiar abstraction of a network topology to each module. A network object consists of an abstract topology, as well as a policy function applied to the abstract topology. For example, the abstract topology could be a subgraph of the real topology, one big virtual switch spanning the entire physical network, or anything in between. The abstract topology may consist of a mix of physical and virtual switches, and may have multiple levels of nesting on top of the one real network. To illustrate how topology abstraction may help in creating modular SDN applications we look at two examples: a "many-to-one" mapping in which several physical switches are made to appear as one virtual switch and a "one-to-many" mapping in which one physical switch is presented as several virtual switches.

Many-to-one. While MAC-learning is an effective way to learn the locations of hosts in a network, the need to compute spanning trees makes Ethernet protocols unattractive in large networks. Instead, a programmer could combine MAC-learning at the edge of the network with shortest-path routing (for unicast traffic) and multicast trees (for broadcast and flooding traffic) in the network interior [18, 23]. Topology abstraction provides a simple way to realize this functionality, as shown in Figure 2. The MAC-learning module sees the network as one big switch `V`, with one port for each edge link in the underlying physical network (dotted lines). The module can run the conventional MAC-learning program to learn where hosts are located. When a previously-unknown host sends a packet, the module associates the source address with the input port, allowing the module to direct future traffic destined to this address out that port. When switch `V` receives a packet destined to an unknown address, the module floods the packet; otherwise, the switch forwards the traffic to the known output port.

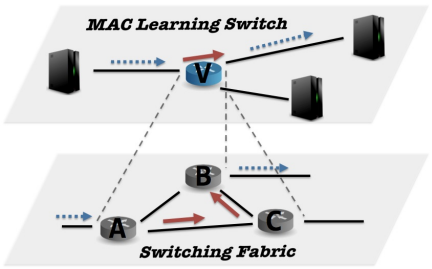


Figure 2: Many physical switches to one virtual.

The "switching fabric" of switch `V` is implemented by the switching-fabric module, which sees the entire physical network. the switching-fabric module performs routing from *one edge link to another* (e.g., from the ingress port at switch `A` to the egress port at switch `B`), This requires some coordination between the two modules, so the MAC-learner can specify the chosen output port(s), and the switching-fabric module can direct traffic on a path to the egress port(s).

As a general way to support coordination, we introduce an *abstract packet model*, incorporating the concept of *virtual packet headers* that a module can push, pop, and inspect, just like the actions OpenFlow supports on real packet-header fields like VLAN tags and MPLS labels. When the MAC-learning module directs traffic from an input port to an output port, the switching-fabric module sees traffic with a virtual packet header indicating the corresponding ingress and egress ports in its view of the network. A run-time system can perform the necessary mappings between the two abstract topologies, and generate the appropriate rules to forward traffic from the ingress port to the appropriate egress port(s). In practice, a run-time system may represent virtual packet-header fields using VLAN tags or MPLS labels, and in-

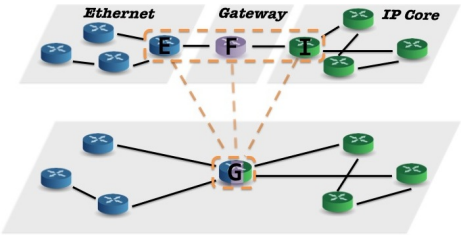


Figure 3: One physical switch to many virtual.

stall rules that push, pop, and inspect these fields.

One-to-many. Enterprise networks often consist of several Ethernet islands interconnected by gateway routers to an IP core, as shown in Figure 3. To implement this behavior, an SDN programmer would have to write a single, monolithic program that handles network events differently depending on the role the switch is playing in the network. This program would implement MAC-learning and flooding to unknown destinations for switches within Ethernet islands, shortest-path routing on IP prefixes for switches in the IP core, and gateway logic for devices connecting an island to the core. The gateway logic would be complicated, as the switch would need to act simultaneously as MAC-learner, IP router, and MAC-rewriting repeater and ARP server.

A better alternative would be to implement the Ethernet islands, IP core, and gateway routers using separate modules operating on a subset of the topology, as shown in Figure 3. This design would allow the gateway router to be decomposed into three virtual devices: one in the Ethernet island (`E`), another in the IP core (`I`), and a third interconnecting the other two (`F`). Likewise, its logic could be decomposed into three orthogonal pieces: a MAC-rewriting repeater that responds to ARP queries for its gateway address (on `F`), an Ethernet switch (on `E`), and an IP router (on `I`). The programmer would write these modules separately and rely on a run-time system to combine them into a single program.

For example, suppose a host in the Ethernet LAN sends a packet to a destination reachable via the IP core. In the Ethernet LAN, this packet has a destination MAC address of the gateway. The Ethernet module would generate a rule in switch `E` that matches traffic destined to the gateway's MAC address and forwards out `E`'s right port. The gateway module would generate a rule in switch `F` that matches packets from `F`'s left port destined to the gateway's MAC address and, after rewriting MAC headers appropriately, forwards out `F`'s right port. The IP core module would generate rules in switch `I` that match packets based on the destination IP address to forward traffic to the next hop along a path to the destination. A run-time system can combine these three sets of rules to generate the rules for the physical gateway switch `G`. Switch `G` would match traffic entering on its left two ports based on the gateway's destination MAC address and the destination IP address to forward via one of the two right ports, as chosen by the IP core module.

	Conventional SDN	Pyretic
Packet	Fixed OF fields	Extensible stacks of values
Policy	Prioritized OF rules	Functions of located packets
Network	One concrete network	Network object hierarchies

Table 1: Pyretic abstraction in three dimensions

3 The Pyretic Programming Language

Any SDN platform needs a model of data packets, forwarding policies, and the network that applies these policies—as summarized in Table 1. Compared to conventional platforms [7, 12, 2, 3, 19], our Pyretic language raises the level of abstraction by introducing an abstract packet model (Section 3.1), an algebra of high-level policies (Section 3.2), and network objects (Section 4).

3.1 Abstract Packet Model

The heart of the Pyretic programming model is a new, extensible packet model. Conceptually, each packet flowing through the network is a *dictionary* that maps field names to values. These fields include entries for (1) the packet location (either physical or virtual), (2) standard OpenFlow headers (e.g., source IP, destination IP, source port, etc.), and (3) custom data. The custom data is housed in *virtual fields* and is not limited to simple bit strings—a virtual field can represent an arbitrary data structure. Consequently, this representation provides a general way to associate high-level information with packets and enable coordination between modules.

In addition to extending the *width* of a packet by including virtual fields, we also extend its *height* by allowing every field (including non-virtual ones) to hold a *stack* of values instead of a single bitstring. These stacks allow Pyretic to present the illusion of a packet travelling through *multiple* levels of abstract networks. For example, to "lift" a packet onto a virtual switch, the run-time system *pushes* the location of the virtual switch onto the packet. Having done so, that virtual switch name sits on top of the concrete switch name. When a packet leaves a virtual switch, the run-time system *pops* a field off the appropriate stack. In the example in Figure 2, this enables the MAC-learning module to select an egress port on virtual switch `V` without knowing about the existence of switches `A`, `B`, and `C` underneath.

Expanding on the example in Figure 2, consider a packet `p` entering the network at physical switch `A` and physical input port `3`. We can represent `p` as:

```
{switch: A, inport: 3, vswitch: V, ... }
```

Pushing virtual switch name `V` on to the switch field of `p` produces a new packet with `V` on top of `A`:

```
{switch: [V, A], inport: 3, ... }
```

The push above hides the identity of the physical switch `A` and reveals only the identity of the virtual switch `V` to observers that only examine top-most stack values. This mechanism allows Pyretic applications to hide the concrete network and replace it with a new abstract one.

Thus far, we have experimented primarily with the abstraction of location information: switches and ports. However, there is nothing special about those fields. We can virtualize IP addresses, MAC addresses or any other information in a packet, if an application demands it. Programs must maintain the invariant that the standard OpenFlow header fields do not contain the empty stack; packets with an empty stack in such a field will not be properly realized as a standard OpenFlow packet.

Ideally, OpenFlow switches would support our extended packet model directly, but they do not. Our runtime system is responsible for bridging the gap between the abstract model and the OpenFlow-supported packets that traverse the network. It does so by generating a unique identifier that corresponds to a unique set of non-OpenFlow-compliant portions of the packet (i.e., all virtual fields and everything but the top of the stack in an OpenFlow-compliant field). This identifier is stored in spare bits in the packet.[2] Our run-time system manages a table that stores the mapping between unique ids and extended data. Hence, programmers do not need to manage this mapping themselves and can instead work in terms of high-level abstractions.

3.2 High-Level Policy Functions

Pyretic contains a sublanguage for specifying *static* (i.e., unchanging) policies. A static policy is a "snapshot" of a network's global forwarding behavior, represented as an abstract function from a located packet to a multiset of located packets. The output multiset may be empty; if so, the policy has effectively dropped the input packet. The output multiset may contain a single packet at a new location (e.g., unicast)—typically, though not always, an output port on the other side of the switch. Finally, the output multiset may contain several packets (e.g., multicast or broadcast). Of course, one cannot build many useful network applications with just a single static, unchanging policy. To do so, one must use a *series* of static policies (i.e., a *dynamic* policy).

3.2.1 Static Policy Functions

We first describe the details of the static policy language, which we call NetCore.[3] NetCore contains several distinct elements including *actions* (the basic packet-processing primitives), *predicates* (which are used to select certain subsets of packets), *query policies* (which are used to observe packets traversing the network), and finally *policy combinators*, which are used to mix primitive actions, predicates, and queries together to craft sophisticated policies from simple components. Figure 4 summarizes the syntax of the key elements of NetCore.

Primitive Actions:

A	::=	`drop` | `passthrough` | `fwd(port)` | `flood` | `push(h=v)` | `pop(h)` | `move(h1=h2)`

Predicates:

P	::=	`all_packets` | `no_packets` | `match(h=v)` | `ingress` | `egress` | *P* & *P* | (*P* | *P*) | ~*P*

Query Policies:

Q	::=	`packets(limit,[h])` | `counts(every,[h])`

Policies:

C	::=	*A* | *Q* | *P*[*C*] | (*C* | *C*) | *C* >> *C* | `if_`(*P*,*C*,*C*)

Figure 4: Summary of static NetCore syntax.

Primitive actions. Primitive actions are the central building blocks of Pyretic policies; an action receives a located packet as input and returns a set of located packets as a result. The simplest is the `drop` action, which produces the empty set. The `passthrough` action produces the singleton set `{p}` where `p` is the input packet. Hence `passthrough` acts much like an identity function—it does not even move the packet from its input port. Perhaps surprisingly, `passthrough` is quite useful in conjunction with other policies and policy combinators. On input packet `p`, the `fwd(port)` action produces the singleton set containing the packet relocated to outport `port` on the same switch as a result. The `flood` action sends packets along a minimum spanning tree, excepting the incoming interface[4]. When viewed as a function, `flood` receives any packet located at an inport on switch `s` and produces an output set with one copy of the packet at each outport on `s` that belongs to a minimum spanning tree for the network (maintained by the run-time system). The last three actions, `push`, `pop`, and `move`, each yield a singleton set as their output: `push(h=v)` pushes value `v` on to field `h`; `pop(h)` pops a value off of field `h`; and `move(h1=h2)` pops the top value on field `h2` and pushes it on to `h1`.

Predicates. Predicates are essential for defining policies (or parts of policies) that act only on a *subset* of packets traversing the network. More specifically, given an input packet `p`, the policy `P[C]`, applies the policy function `C` to `p` if `p` satisfies the predicate `P`. If `p` does not satisfy `P` then the empty set is returned. (In other words, the packet is dropped.) Predicates include `all_packets` and `no_packets`, which match all or no packets, respectively; `ingress` and `egress` which, respectively, match any packets entering or exiting the net-

[2] Any source of spare bits (e.g., MPLS labels) could be used. Our current implementation uses the VLAN field.

[3] This variant of NetCore is an extension and generalization of a language with the same name, described in our earlier work [14].

[4] The same definition used by Openflow for its `flood` action.

work; and `match(h=v)`, matching all packets for which value `v` is the top value on the stack at field `h`. Complex predicates are constructed using basic conjunction (`&`), disjunction (`|`), and negation (`~`) operators. The form `match(h1=v1,h2=v2)` is an abbreviation for `match(h1=v1) & match(h2=v2)`.

As an example, the policy `flood`, on its own, will broadcast every single packet that reaches *any* inport of *any* switch *anywhere* in the network. On the other hand, the policy

```
match(switch=s1,inport=2,srcip='1.2.3.4') [flood]
```

only broadcasts those packets reaching `switch s1`, `inport 2` with source IP address `1.2.3.4`. All other packets are dropped.

Policies. Primitive actions `A` are policies, as are restricted policies `P[C]`. NetCore also contains several additional ways of constructing policies.

As discussed in Section 2, sequential composition is used to build packet-processing pipelines from simpler components. Semantically, we define sequential composition `C1 >> C2` as the function `C3` such that:

```
C3(packet) = C2(p1) ∪ ... ∪ C2(pn)
  when {p1,...,pn} = C1(packet)
```

In other words, we apply `C1` to the input, generating a set of packets (`p1`,..., `pn`) and then apply `C2` to each of those results, taking their union as the final result.

As an example, consider the following policy, which modifies the destination IP of any incoming packet to 10.0.0.1 and forwards the modified packet out port 3.

```
pop(dstip) >> push(dstip='10.0.0.1') >> fwd(3)
```

Indeed, the modification idiom is common enough that we define an abbreviation for it:

```
modify(h=v) = pop(h) >> push(h=v)
```

As a more elaborate example, consider a complex policy `P2`, designed for forwarding traffic using a set of tags (`staff`, `student`, `guest`) stored in a virtual field named `USERCLASS`. Now, suppose we would like to apply the policy for `staff` to a particular subset of the traffic arriving on network. To do so, we may write a policy `P1` to select and tag the relevant traffic. To use `P1` and `P2` in combination, we exploit sequential composition: `P1 >> P2`. Such a program is quite modular: if a programmer wanted to change the forwarding component, she would change `P2`, while if she wanted to change the set of packets labeled `staff`, she would change `P1`.

Parallel composition is an alternative and orthogonal form of composition to sequential composition. The parallel composition `P3 | P4` behaves as if `P3` and `P4` were executed on every packet simultaneously. In other words, given an input packet `p`, `(P3 | P4)(p)` returns the set of packets `P3(p) U P4(p)`.

Continuing our example, if a programmer wanted to apply the policy `P3 | P4` to packets arriving at switch `s1` and a different policy `P6` to packets arriving at `s2`, she could construct the following composite policy `P7`.

```
P5 = P3 | P4
P6 = ...
P7 = match(switch=s1)[P5] | match(switch=s2)[P6]
```

After recognizing a security threat from source IP address, say address `1.2.3.4`, the programmer might go one step further creating policy `P8` that restricts `P7` to applying only to traffic from other addresses (implicitly dropping traffic from `1.2.3.4`).

```
P8 = ~match(srcip='1.2.3.4')[P7]
```

The policy `if_` is a convenience conditional policy. For example, if the current packet satisfies `P`, then

```
if_(P, drop, passthrough)
```

drops that packet while leaving all others untouched (allowing a subsequent policy in a pipeline of sequential compositions to process it). Conditional policies are a derived form that can be encoded using parallel composition, restriction, and negation.

Queries. The last kind of policy we support is a *query policy* (`Q`). Intuitively, a query is an abstract policy that directs information from the physical network to the controller platform. When viewed as a function, a query receives located packets as arguments and produces new located packets as results like any other policy. However, the resulting located packets do not find themselves at some physical port on some physical switch in the network. Instead, these packets are diverted to a data structure resident on the controller called a "bucket".

NetCore contains two queries: `counts` and `packets`, which, abstractly, direct packets to two different types of buckets, a `packet_bucket` and a `counting_bucket`, respectively. Applications register listeners (i.e., callbacks) with buckets; these callbacks are invoked to process the information contained in the bucket. Semantically, the two query policies differ only in terms of the information each bucket reveals to its listeners.

The `packet_bucket`, as its name suggests, passes entire packets to its listeners. For example, `packets(limit,['srcip'])` invokes its listeners on up to `limit` packets for each source IP address. The two most common values for `limit` are `None` and 1, with `None` indicating the bucket should process an unlimited number of packets. More generally, a list of headers is allowed: the bucket associated with `packets(1,[h1,...,hk])` invokes each listener on at most 1 packet for each distinct record of values in fields `h1` through `hk`.

The `counting_bucket` supplies its listeners with aggregate packet statistics, not the packets themselves. Hence, it may be implemented using OpenFlow counters in switch hardware. The policy

```
from pyretic.lib import *

def main():
  return flood
```

Figure 5: A complete program: `hub.py`.

`counts(every,['srcip'])` creates a bucket that calls its listeners every `every` seconds and provides each listener with a dictionary mapping source IP addresses to the cumulative number of packets containing that source IP address received by the bucket. As above, counting policies may be generalized to discriminate between packets on the basis of multiple headers: `counts(every,[h1,...,hk])`.

A query policy `Q` may be used in conjunction with any other policy we define. For example, if we wish to analyze traffic generated from IP address `1.2.3.4` using `Q`, we may simply construct the following.

```
match(srcip='1.2.3.4') [Q]
```

If we wanted to both forward and monitor a certain subset of packets, we use parallel composition as before:

```
match(srcip='1.2.3.4') [Q | fwd(3)]
```

3.2.2 From Static Policies to Dynamic Applications

After defining a static policy, the programmer may use that policy within a Pyretic application. Figure 5 presents the simplest possible, yet complete, Pyretic application. It imports the Pyretic library, which includes definitions of all primitive actions (such as `flood`, `drop`, etc.), policy operators and query functions, as well as the run-time system. The program itself is trivial: `main` does nothing but return the `flood` policy. A Pyretic program such as this one is executed by starting up a modified version of the POX run-time system [19]. POX reads the Pyretic script and executes `main`. Although we use POX for low-level message processing, our use of POX is not essential. A Pyretic-like language could be built on top of any low-level controller.

Monitoring. Figure 6 presents a second simple application, designed to monitor and print packets from source IP `1.2.3.4` to the terminal. In this figure, the `dpi` function first creates a new packet-monitoring policy named `q`. Next, it registers the `printer` listener with the query `q` using `q`'s `when` method. This listener will be called each time a packet arrives at the `packet_bucket` to which `q` forwards. Finally, the `dpi` function constructs and returns a policy that embeds the query within it. The `main` function uses `dpi` and further composes it with a routing policy (the simple `flood`).

MAC-learning. Figure 7 presents an MAC-learning module that illustrates how to construct a dynamic policy. It is designed assuming that network hosts do not move. It initially operates by flooding all packets it receives. For each switch, when a packet with a new source MAC address (say, MAC address `M`) appears at one of its input ports (say, inport `I`), it concludes that `M` must live off `I`. Consequently, it refines its forwarding behavior so that packets with destination MAC address `M` are no longer flooded, but instead forwarded only out of port `I`.

```
def printer(pkt):
  print pkt

def dpi():
  q = packets(None,[])
  q.when(printer)
  return match(srcip='1.2.3.4')[q]

def main():
  return dpi() | flood
```

Figure 6: Deep packet inspection.

```
def learn(self):

  def update(pkt):
    self.P =
      if_(match(dstmac=pkt['srcmac'],
                switch=pkt['switch']),
          fwd(pkt['inport']),
          self.P)

  q = packets(1,['srcmac','switch'])
  q.when(update)

  self.P = flood | q

def main():
  return dynamic(learn)()
```

Figure 7: MAC-learning switch.

Examining a few of the details of Figure 7, we see that the last line of the `learn` function is the line that initializes the policy—it starts out flooding all packets and using `q` to listen for packets with new source MAC addresses. The listener for query is the function `update`, which receives packets with new source MACs as an argument. That listener updates the dynamic policy with a conditional policy that tests future packets to see if their destination MAC is equal to the current packet's source MAC. If so, it forwards the packet out the inport on which the current packet resides. If not, it invokes the existing policy `self.P`. In this way, over time, the policy is extended again and again until the locations of all hosts have been learned.

The last line of Figure 7 uses the function `dynamic` to wrap up `learn` and produce a new dynamic policy class, whose constructor it then calls to produce a operational dynamic policy instance.

Load balancer. As a final example in this section, we show how to construct a simple dynamic server load bal-

ancer. Doing so illustrates a common Pyretic programming paradigm: One may develop dynamic applications by constructing parameterized static policies, listening for network events, and then repeatedly recomputing the static policy using different parameters as the network environment changes. The example also illustrates the process of building up a somewhat more complex policy by defining a collection of ordinary Python functions that compute simple, independent components of the policy, which are subsequently composed together.

Figure 8 presents the code, which spreads requests for a single public-facing IP address over multiple back-end servers. The first three functions in the figure collaborate to construct a static load balancer. In the first function (`subs`), variable `c` is the client IP address prefix, `p` is the service's public-facing IP address, and `r` is the specific replica chosen. This function rewrites any packet whose source IP address matches `c` and destination IP address is `p` so the destination IP address is `r`, and vice versa. All other packets are left unmodified. The next function, `rewrite`, iterates `subs` over a dictionary `d` mapping IP prefixes to server replicas. To simplify this code, we have overloaded sequential composition (`>>`) so that it can be applied to a list of policies (placing each element of the list in sequence). Hence, intuitively, `rewrite` sends each packet through a pipeline of tests and when the test succeeds, the packet is transformed. The third function, `static_lb` invokes `rewrite` with a function `balance` (definition omitted from the figure) that partitions possible clients and assigns them to replicas, using a lists of server replicas `R` and a dictionary `H` containing a history of traffic statistics.

Now, to build a dynamic load balancer that changes the mapping from clients to server replicas over time, consider `lb`. This dynamic balancer issues a query `q` that computes a dictionary mapping source IP addresses to packet counts every minute (60 seconds). Each time the query returns a new `stats` value, the policy invokes `rebalance`, which updates the history `H` and recomputes a new load balancing policy using `static_lb`.

```
def subs(c,r,p):
  c_to_p = match(srcip=c,dstip=p)
  r_to_c = match(srcip=r,dstip=c)
  return c_to_p[modify(dstip=r)] |
         r_to_c[modify(srcip=p)] |
         (~r_to_c & ~c_to_p)[passthrough]

def rewrite(d,p):
  return (>>)([subs(c,r,p) for c,r in d])

def static_lb(p,R,H):
  return rewrite(balance(R,H),p)

def lb(self,p,R,H):

  def rebalance(stats):
    H = H.update(stats)
    self.P = static_lb(p,R,H)

  q = counts(60,['srcip'])
  q.when(rebalance)
  self.P = static_lb(p,R,H) | match(dstip=p)[q])
```

Figure 8: Dynamic load balancer (excerpts).

4 Network Objects

The policy language described in the preceding section provides programmers with flexible constructs that make it easy to build sophisticated network applications out of simple, independent components. However, it suffers from a significant limitation: programmers must specify policies in terms of the underlying physical topology. This hinders code reuse since policies written for one topology typically cannot be used with other topologies.

To address this limitation, Pyretic also provides *network objects* (or simply *networks*), which allow programmers to abstract away details of the physical topology and write policies in terms of abstracted views of that network—providing an important form of modularity. We do this by allowing a new *derived* network to be built on top of an already existing *underlying* network (and, as implied by this terminology, Pyretic programmers may layer one derived network atop another).

Each network object has three key elements: a topology, a policy, and, for derived networks, a *mapping*. The topology object is simply a graph with switches as nodes and links as edges. The policy specifies the intended behavior of the network object with respect to that topology. The mapping comprises functions establishing an association between elements of the derived topology and those of its underlying topology.

The *base* network object represents the physical network. The Pyretic run-time system implements a discovery protocol that learns the physical topology using a combination of OpenFlow events and simple packet probes. The run-time system ultimately resolves all derived network policies into a single policy that can be applied to the base network.

A derived network object's mapping comprises the following functions:

- A function to map changes to the underlying topology up to changes on the derived topology, and
- A function to map policies written against the derived topology down to a semantically equivalent policy expressed only in terms of the underlying topology.

Pyretic provides several constructs for implementing these functions automatically. In most situations, the programmer simply specifies the mapping between elements of the topologies, along with a function for calculating forwarding paths through the underlying topology, and Pyretic calculates correct implementations automati-

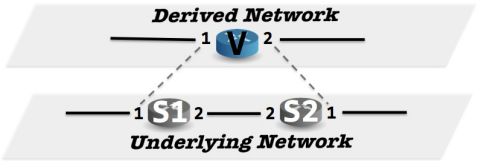


Figure 9: Derived "big switch" topology.

cally. The next few paragraphs describe these features in detail. For concreteness, we consider a running example where the derived network contains a single "big switch" as shown in Figure 9.

Transforming Topologies. The first step involved in implementing the "big switch" is to specify the relationship between elements of the underlying topology and elements of the derived topology. In this case, because the derived topology only contains a single switch, the association between underlying and derived topology elements can be computed automatically. To represent this association, we will use a dictionary that maps switch-port pairs `(vswitch,vport)` in the derived network to switch-port pairs `(switch,port)` in the underlying network. The following Python function computes a dictionary that relates ports on the switch in the derived network to ports at the perimeter of the underlying network:

```
def bfs_vmap(topo):
  vswitch = 1
  vport = 1
  for (switch, port) in topo.egress_locations:
      vmap[(vswitch, vport)] = (switch, port)
      vport += 1
  return vmap
```

Using this dictionary, it is straightforward to build the graph representing the derived topology—it consists of a single switch with the ports specified in the domain of the dictionary. Likewise, it is straightforward to map changes to the underlying topology up to changes for the derived topology. Pyretic provides code to implement these functions automatically from the `vmap` dictionary.

Transforming Policies. The next step involved in implementing the "big switch" is to transform policies written for the derived network into policies for the underlying network. This turns out to be significantly more challenging because it involves implementing a policy written against one topology, using the switches and links provided in a completely different topology. However, Pyretic's abstract packet model and support for sequential composition allow the transformation to be expressed in a clean and elegant way.

The transformation uses three auxiliary policies:[5]

- *Ingress Policy:* "lifts" packets in the underlying network up into the derived network by pushing appropriate switch and port identifiers onto the stack of values maintained in Pyretic's abstract packet model.
- *Egress policy:* "lowers" packets from the derived network to the underlying network by popping the switch and port identifier from the stack of values maintained in Pyretic's abstract packet model.
- *Fabric policy:* implements forwarding between adjacent ports in the derived network using the switches and links in the underlying topology. In general, calculating this policy involves computing a graph algorithm on the underlying topology.

[5] As before, Pyretic can automatically generate these from a vmap.

```
def virtualize(ingress_policy,
               egress_policy,
               fabric_policy,
               derived_policy):
  return if_(~match(vswitch=None),
    (ingress_policy >>
     move(switch=vswitch,
          inport=vinport) >>
     derived_policy >>
     move(vswitch=switch,
          vinport=inport,
          voutport=outport)),
    passthrough) >>
  fabric_policy >>
  egress_policy
```

Figure 10: Virtualization transformation.

With these auxiliary policies, the policy transformation can be expressed by composing several policies in sequence: ingress policy, derived policy, fabric policy, and egress policy. Figure 10 defines a general function `virtualize` that implements this transformation.

Example. To illustrate ingress, egress, and fabric policies, consider a specific physical topology consisting of two switches `S1` and `S2`, each with an outward-facing port and connected to each other by a link as shown in Figure 9. The policy running on the derived switch encodes the behavior of a repeater hub, as shown in Figure 5. The ingress policy is as follows:

```
ingress_policy =
( match(switch=S1, inport=1)
   [push(vswitch=V, vinport=1)]
| match(switch=S2, inport=1)
   [push(vswitch=V, vinport=2)])
```

It simply pushes the derived switch `V` and inport onto the corresponding "virtual" header stacks. The egress policy is symmetric:

```
egress_policy = match(vswitch=V)
  [if_(match(switch=S1, voutport=1)
    | match(switch=S2, voutport=2),
    pop(vswitch, vinport, voutport),
    passthrough)]
```

It pops the derived switch, inport, and outport from the appropriate virtual header stacks if the switch is labeled

with derived switch V, and otherwise passes the packet through unmodified. The fabric policy forwards packets labeled with derived switch V along the (unique) path between S1 and S2:

```
fabric_policy = match(vswitch=V)[
 ( match(switch=S1, voutport=1)[fwd(1)]
 | match(switch=S1, voutport=2)[fwd(2)]
 | match(switch=S2, voutport=1)[fwd(2)]
 | match(switch=S2, voutport=2)[fwd(1)])]
```

To illustrate these definitions, consider processing of a packet with the following headers.

```
{ switch:S1, inport:1, ... }
```

Recall that Pyretic's packet model treats the location of the packet as a header field. The first step of processing checks whether the packet has already entered the derived network, by testing for the presence of the vswitch header field. In this case, the packet does not contain this field so we treat it as being located at the ingress port of the derived switch and take the first branch of the conditional in Figure 10. Next, we evaluate ingress_policy which, by the first disjunct, pushes the headers vswitch=V and vinport=1 onto the packet, yielding a packet with the following headers:

```
{ switch:S1, inport:1,
  vswitch:V, vinport:1, ... }
```

Next we move the vswitch and vinport headers to switch and inport, and evaluate the policy written against the derived network (here simply flood). Flooding the packet on the derived network, generates a packet on outport 2 in this case:

```
{ switch:[V, S1], inport:[1, 1],
  outport:2, ...}
```

We then move the switch, inport, and outport headers to the corresponding virtual header stacks, which has the effect of restoring the original switch and inport headers,

```
{ switch:S1, inport:1,
  vswitch:V, vinport:1, voutport:2 }
```

and evaluate the fabric policy, which forwards the packet out port 2 of switch S1. Finally, the egress policy passes the packet through unmodified and the underlying topology transfers the packet to port 2 on switch S2:

```
{ switch:S2, inport:2,
  vswitch:V, vinport:1, voutport:2 }
```

This completes the first step of processing on the physical network. In the second step of processing, the packet already has virtual switch, inport, and outport labels. Hence, we do not calculate virtual headers as before and instead skip straight to the fabric policy, which forwards the packet out port 1 of S2. Now the packet does satisfy the condition stated in the egress policy, so it pops the virtual headers and forwards the packet out to its actual destination.

5 Example Pyretic Applications

To experiment with the Pyretic design, we have implemented a collection of applications. Table 2 lists a selection of these examples, highlighting the key features of Pyretic utilized, where the examples are discussed in the paper, and corresponding file names in the reference implementation [1]. Most terms in the features column should be familiar from prior sections. The term "novel primitives" simply refers to basic, but novel, features of Pyretic such as passthrough policies. Due to space constraints, we have omitted discussion of certain advanced Pyretic features that are needed to implement some applications including *traffic generation*, *topology-aware predicates*, *dynamic nesting*, and *recursion*. Section 5.1 elaborates on some of the additional applications found in our reference implementation and the key features they use. Section 5.2 concludes by presenting a "kitchen sink" example that utilizes all of Pyretic's features to write a truly modular application in just a few lines of code.

5.1 Pyretic Example Suite

ARP. The ARP application demonstrates how a Pyretic program can inject new packets into the network, and thereby respond to ARP traffic on behalf of hosts.

Firewalls. The firewall applications construct stateless (static) and stateful (dynamic) firewalls. These applications are similar in nature to the load balancer described in Section 3.2.2, but go a step further by demonstrating an advanced technique we call *dynamic nesting* in which one dynamic policy includes another dynamic policy within it. These firewalls also exploit topology-aware predicates such as ingress (which identifies packets at the network ingress) and egress (which identifies packets at the network egress).

Gateway. The gateway example implements the picture in Figure 3. The physical topology consists of three parts: an Ethernet island (switches 2, 3, and 4), a gateway router (switch 1), and an IP core (switches 5, 6, and 7). The gateway router is responsible for running several different pieces of logic. It is difficult to reuse standard components when all modules must share the same physical switch, so we abstract switch 1 to three switches (1000, 1001, 1002)—running MAC-learning, gateway logic, and IP routing, respectively. Unlike previous examples, the ports and links connecting these three switches are *completely virtual*—that is they map to no physical port, even indirectly. We encapsulate these components into a network object named GatewayVirt that performs the mechanical work of copying the base object and modifying it accordingly.

To a first approximation, here is how each of the virtu-

Examples	Pyretic Features Used	Section	File
Hub	static policy	3.2.2	`hub.py`
MAC-Learning	dynamic policy; queries; parallel comp.	3.2.2	`mac_learner.py`
Monitoring	static policy; queries; parallel comp.	3.2.2	`monitor.py`
Load Balancers	static policy; queries; parallel & sequential comp.; novel primitives	3.2.2	`load_balancer.py`
Firewalls	dynamic policy; queries; parallel & sequential comp.; novel primitives; topology-aware predicates; dynamic nesting	-	`firewall.py`
ARP	static policy; queries; parallel comp.; traffic generation	-	`arp.py`
Big Switch	static policy; topology abstraction; virtual headers; parallel comp.; novel primitives	4	`bfs.py`
Spanning Tree	static policy; topology abstraction; virtual headers; parallel comp.; novel primitives	-	`spanning_tree.py`
Gateway	static policy; topology abstraction; virtual headers; recursion; parallel & sequential comp.; novel primitives	-	`gateway.py`
Kitchen Sink	dynamic policy; topology abstraction; virtual headers; parallel comp.; novel primitives; topology-aware predicates; dynamic nesting	5.2	`kitchen_sink.py`

Table 2: Selected Pyretic examples.

alization components are implemented:[6]

- *Ingress policy*: Incoming packets to the physical gateway switch from the Ethernet side are tagged with `vswitch=1000` (and the appropriate `vinport`), and incoming packets to the physical gateway switch from the IP side are tagged with `vswitch=1002` (and the appropriate `vinport`).
- *Fabric policy*: For switches `1000-1002`, the fabric policy modifies the packet's virtual headers, effectively "moving" the packet one-step through the chain of switches. When moving the packet to a virtual port, the fabric policy recursively applies the entire policy (including ingress, fabric, and egress policies). The recursion halts when the packet is moved to a non-completely virtual port, at which time the packet is forwarded out of the corresponding physical port.
- *Egress policy*: As virtual links span at most one physical link, we strip the virtual headers after each forwarding action on the base network.

5.2 Putting it All Together

We conclude with an example application addressing the motivating "many-to-one" scenario discussed in Section 2.2 and shown in Figure 3. We implement the functionality of the Ethernet island by handling ARP traffic using the corresponding module from Table 2 and all other traffic with the familiar MAC-learning module.

```
eth = if_(ARP,dynamic(arp)(),dynamic(learn)())
```

We take the load balancer from Section 3.2.2 and combine it with a dynamic firewall from the examples table. This firewall is written in terms of white-listed traffic from client IPs to public addresses, creating another interesting twist— easily solved using Pyretic's operators. Specifically, the correct processing order of load balancer and firewall turns out to be *direction-dependent*. The firewall must be applied before load balancing for incoming traffic from clients—as the firewall must consider the original IP addresses, which are no longer be available after the load balancer rewrites the destination address that of a replica. In the other direction, the load balancer must first restore the original IP address before the firewall is applied to packets returning to the client.

```
fwlb = if_(from_client, afw >> alb, alb >> afw)
```

Finally, we complete our IP core by taking this combined load balancer/firewall and sequentially composing it with a module that implements shortest-path routing to the appropriate egress port—by running MAC-learning on a shortest path big switch!

```
ip = fwlb >> virtualize(dynamic(learn)(),
                        BFS(ip_core) )
```

The final component is our gateway logic itself. The gateway handles ARP traffic, rewrites source and destination MAC addresses (since these change on subnet transitions), and forwards out the appropriate port.

```
gw = if_(ARP,dynamic(arp)(),
             rewrite_macs(all_macs) >>
             ( eth_to_ip[fwd(2)] |
               ip_to_eth[fwd(1)] ))
```

We can then combine each of these policies, restricted to the appropriate set of switches, in parallel and run on the virtualized gateway topology discussed previously.

```
virtualize(in_(ethernet)[ eth ] |
           in_(gateway)[  gw  ] |
           in_(ip_core)[  ip  ],
           GatewayVirt(Recurse(self))
```

The `virtualize` transformation from Section 4 generates the ultimate policy that is executed on the base network.

6 Related Work

In recent years, SDN has emerged as an active area of research. There are now a number of innovative controller platforms based on the OpenFlow API [13] that make it possible to manage the behavior of a network using general-purpose programs [7, 12, 19, 2, 3, 24, 21]. Early

[6] See the reference implementation [1] for further details.

controller platforms such as NOX [7] offered a low-level programming interface based on the OpenFlow API itself; recent controllers provide additional features such as composition, isolation, and virtualization. We briefly review related work in each of these areas.

Composition. Pyretic's parallel and sequential composition operators resemble earlier work on the Click modular router [11]. However, rather than targeting multiple packet-processing modules within a single software router, Pyretic focuses on composing the control logic that affects the handling of traffic on an entire network of OpenFlow switches. Other controller platforms with some support for composition include Maestro [3], which allows programmers to write applications in terms of user-defined views of network state, and FRESCO [22], which provides a high-level language for defining security policies. Pyretic is distinguished from these systems in modeling policies as mathematical functions on packets, and in providing direct support for policy composition. In particular, unlike previous systems, Pyretic's composition operators do not require programmers to resolve conflicts by hand.

Isolation. To support multiple applications executing simultaneously in a single network, several controllers now support network "slices". Each slice may execute a different program while the controller provides isolation. One popular such controller is FlowVisor [21], a hypervisor that enforces strict traffic isolation between the controllers running on top of it, and also manages provisioning of shared resources such as bandwidth and the controller itself. Another recent proposal uses an extension to the NetCore compiler to provide traffic isolation by construction [8]. Finally, controllers that support the creation of virtual networks typically provide a form of isolation [16]. Pyretic's composition operators—in particular, sequential composition—make it straightforward to implement network slices. In addition, unlike controllers that only provide strict slicing, Pyretic can also be used to decompose a single application into different modules that affect the same traffic.

Network Objects. Pyretic's network objects generalize the global network views provided in other SDN controllers, such as NOX [7], ONIX [12], and POX [19]. In these systems, the network view (or network information base) represents the global topology as an annotated graph that can be configured and queried. Some systems go a step further and allow programmers to define a mapping between a representation of the physical topology and a simplified representation of the network. For example, the "big switch" abstraction [16, 4, 5, 20] can greatly simplify the logic of applications such as access control and virtual machine migration. Pyretic's network objects can be used to implement a wide range of abstract topologies. Moreover, as described in Section 4, sequential composition and virtual headers provide a simple and elegant mechanism for building derived network objects that implement a variety of abstract topologies. This mechanism is inspired by a technique for implementing virtual networks originally proposed by Casado et al. [4].

Programming Languages. This paper is part of a growing line of research on applying programming-language techniques to SDN [9, 24, 15, 6, 14]. Our early work on Frenetic [6, 14] introduced a functional language supporting parallel composition and SQL-like queries. This work goes further, by introducing an abstract packet model, sequential composition operator, and topology abstraction using network objects, as well as a new imperative implementation in Python. Taken together, these features facilitate "programming in the large" by enabling programmers to develop SDN applications in a modular way.

7 Conclusion

We believe the right level of abstraction for programmers is not a low-level interface to the data-plane hardware, but instead a higher-level language for writing and composing modules. Pyretic is a new language that allows SDN programmers to build large, sophisticated controller applications out of small, self-contained modules. It provides the programmatic tools that enable network programmers, operators, and administrators to master the complexities of their domain.

Acknowledgments. The authors wish to thank the NSDI reviewers, our shepherd Aditya Akella, Theophilus Benson, Marco Canini, Laurent Vanbever, and members of the Frenetic project for valuable feedback on earlier versions of this paper. This work was supported in part by the NSF under grants 0964409, 1111520, 1111698, 1117696 and TRUST; the ONR under award N00014-12-1-0757; the NSF and CRA under a Computing Innovations Fellowship; and a Google Research Award.

References

[1] Pyretic Reference Implementation. `http://www.frenetic-lang.org/pyretic`.

[2] Beacon: A Java-based OpenFlow Control Platform., Nov. 2010. See `http://www.beaconcontroller.net`.

[3] Cai, Z., Cox, A. L., and Ng, T. S. E. Maestro: A System for Scalable OpenFlow Control. Tech. Rep. TR10-08, Rice University, Dec. 2010.

[4] Casado, M., Koponen, T., Ramanathan, R., and Shenker, S. Virtualizing the Network Forwarding Plane. In *ACM PRESTO* (Nov. 2010).

[5] Corin, R., Gerola, M., Riggio, R., De Pellegrini, F., and Salvadori, E. VeRTIGO: Network Virtualization and Beyond. In *EWSDN* (Oct. 2012), pp. 24–29.

[6] FOSTER, N., HARRISON, R., FREEDMAN, M. J., MONSANTO, C., REXFORD, J., STORY, A., AND WALKER, D. Frenetic: A Network Programming Language. In *ACM ICFP* (Sep. 2011), pp. 279–291.

[7] GUDE, N., KOPONEN, T., PETTIT, J., PFAFF, B., CASADO, M., MCKEOWN, N., AND SHENKER, S. NOX: Towards an Operating System for Networks. *ACM SIGCOMM CCR 38*, 3 (2008).

[8] GUTZ, S., STORY, A., SCHLESINGER, C., AND FOSTER, N. Splendid Isolation: A Slice Abstraction for Software-Defined Networks. In *ACM SIGCOMM HotSDN* (Aug. 2012), pp. 79–84.

[9] HINRICHS, T. L., GUDE, N. S., CASADO, M., MITCHELL, J. C., AND SHENKER, S. Practical Declarative Network Management. In *ACM SIGCOMM WREN* (Aug. 2009), pp. 1–10.

[10] KELLER, E., AND REXFORD, J. The 'Platform as a Service' Model for Networking. In *IMN/WREN* (Apr. 2010).

[11] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK., M. F. The Click Modular Router. *ACM TOCS 18*, 3 (Aug. 2000), 263–297.

[12] 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 *USENIX OSDI* (Oct. 2010), pp. 351–364.

[13] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR 38*, 2 (2008), 69–74.

[14] MONSANTO, C., FOSTER, N., HARRISON, R., AND WALKER, D. A compiler and run-time system for network programming languages. In *ACM POPL* (Jan. 2012), pp. 217–230.

[15] NAYAK, A., REIMERS, A., FEAMSTER, N., AND CLARK, R. Resonance: Dynamic Access Control in Enterprise Networks. In *ACM SIGCOMM WREN* (Aug. 2009), pp. 11–18.

[16] NICIRA. It's time to virtualize the network, 2012. `http://nicira.com/en/network-virtualization-platform`.

[17] NICIRA. Networking in the era of virtualization. `http://nicira.com/sites/default/files/docs/Networking%20in%20the%20Era%20of%20Virtualization.pdf`, 2012.

[18] PERLMAN, R. Rbridges: Transparent Routing. In *IEEE INFOCOM* (Mar. 2004), pp. 1211–1218.

[19] POX. `http://www.noxrepo.org/pox/about-pox/`.

[20] SHENKER, S. The Future of Networking and the Past of Protocols, Oct. 2011. Invited talk at Open Networking Summit.

[21] SHERWOOD, R., GIBB, G., YAP, K.-K., APPENZELLER, G., CASADO, M., MCKEOWN, N., AND PARULKAR, G. Can the Production Network Be the Testbed? In *USENIX OSDI* (Oct. 2010), pp. 365–378.

[22] SHIN, S., PORRAS, P., YEGNESWARAN, V., FONG, M., GU, G., AND TYSON, M. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *Internet Society NDSS* (Feb. 2013). To appear.

[23] TOUCH, J., AND PERLMAN, R. Transparent Interconnection of Lots of Links (TRILL): Problem and Applicability Statement, May 2009. IETF RFC 5556.

[24] VOELLMY, A., AND HUDAK, P. Nettle: Functional Reactive Programming of OpenFlow Networks. In *ACM PADL* (Jan. 2011), pp. 235–249.

[25] WEBB, K. C., SNOEREN, A. C., AND YOCUM, K. Topology switching for data center networks. In *USENIX HotICE* (Mar. 2011).

VeriFlow: Verifying Network-Wide Invariants in Real Time

Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, P. Brighten Godfrey
University of Illinois at Urbana-Champaign
{*khurshi1, xuanzou2, wzhou10, caesar, pbg*}*@illinois.edu*

Abstract

Networks are complex and prone to bugs. Existing tools that check network configuration files and the data-plane state operate offline at timescales of seconds to hours, and cannot detect or prevent bugs as they arise.

Is it possible to *check network-wide invariants in real time*, as the network state evolves? The key challenge here is to achieve extremely low latency during the checks so that network performance is not affected. In this paper, we present a design, VeriFlow, which achieves this goal. VeriFlow is a layer between a software-defined networking controller and network devices that checks for network-wide invariant violations dynamically as each forwarding rule is inserted, modified or deleted. VeriFlow supports analysis over multiple header fields, and an API for checking custom invariants. Based on a prototype implementation integrated with the NOX OpenFlow controller, and driven by a Mininet OpenFlow network and Route Views trace data, we find that VeriFlow can perform rigorous checking within hundreds of microseconds per rule insertion or deletion.

1 Introduction

Packet forwarding in modern networks is a complex process, involving codependent functions running on hundreds or thousands of devices, such as routers, switches, and firewalls from different vendors. As a result, a substantial amount of effort is required to ensure networks' correctness, security and fault tolerance. However, faults in the network state arise commonly in practice, including loops, suboptimal routing, black holes and access control violations that make services unavailable or prone to attacks (e.g., DDoS attacks). Software-Defined Networking (SDN) promises to ease the development of network applications through logically-centralized network programmability via an open interface to the data plane, but bugs are likely to remain problematic since the complexity of software will increase. Moreover, SDN allows multiple applications or even multiple users to program the same physical network simultaneously, potentially resulting in conflicting rules that alter the intended behavior of one or more applications [25].

One solution is to rigorously check network software or configuration for bugs prior to deployment. Symbolic execution [12] can catch bugs through exploration of all possible code paths, but is usually not tractable for large software. Analysis of configuration files [13, 28] is useful, but cannot find bugs in router software, and must be designed for specific configuration languages and control protocols. Moreover, using these approaches, an operator who wants to ensure the network's correctness must have access to the software and configuration, which may be inconvenient in an SDN network where controllers can be operated by other parties [25]. Another approach is to statically analyze snapshots of the network-wide data-plane state [9, 10, 17, 19, 27]. However, these previous approaches operate offline, and thus only find bugs after they happen.

This paper studies the question, *Is it possible to check network-wide correctness in real time as the network evolves?* If we can check each change to forwarding behavior before it takes effect, we can raise alarms immediately, and even prevent bugs by blocking changes that violate important invariants. For example, we could prohibit changes that violate access control policies or cause forwarding loops.

However, existing techniques for checking networks are inadequate for this purpose as they operate on timescales of seconds to hours [10, 17, 19]. [1] Delaying updates for processing can harm consistency of network state, and increase reaction time of protocols with real-time requirements such as routing and fast failover; and processing a continuous stream of updates in a large

[1] The average run time of reachability tests in [17] is 13 seconds, and it takes a few hundred seconds to perform reachability checks in Anteater [19].

network could introduce scaling challenges. Hence, we need some way to perform verification at very high speeds, i.e., within milliseconds. Moreover, checking network-wide properties requires obtaining a view of network-wide state.

We present a design, VeriFlow, which demonstrates that the goal of real-time verification of network-wide invariants is achievable. VeriFlow leverages software-defined networking (SDN) to obtain a picture of the network as it evolves by sitting as a layer between the SDN controller and the forwarding devices, and checks validity of invariants as each rule is inserted, modified or deleted. However, SDN alone does not make the problem easy. In order to ensure real-time response, VeriFlow introduces novel incremental algorithms to search for potential violation of key network invariants — for example, availability of a path to the destination, absence of forwarding loops, enforcement of access control policies, or isolation between virtual networks.

Our prototype implementation supports both OpenFlow [21] version 1.1.0 and IP forwarding rules, with the exception that the current implementation does not support actions that modify packet headers. We microbenchmarked VeriFlow using a stream of updates from a simulated IP network, constructed with Rocketfuel [7] topology data and real BGP traces [8]. We also evaluated its overhead relative to the NOX controller [14] in an emulated OpenFlow network using Mininet [3]. We find that VeriFlow is able to verify network-wide invariants within hundreds of microseconds as new rules are introduced into the network. VeriFlow's verification phase has little impact on network performance and inflates TCP connection setup latency by a manageable amount, around 15.5% on average.

We give an overview of data plane verification and SDN (§ 2) before presenting VeriFlow's design (§ 3), implementation (§ 4), and evaluation (§ 5). We then discuss future (§ 6) and related work (§ 7), and conclude (§ 8).

2 Overview of Approach

VeriFlow adopts the approach of *data plane verification*. As argued in [19], verifying network correctness in the data plane offers several advantages over verifying higher-level code such as configuration files. First, it is closely tied to the network's actual behavior, so that it can catch bugs that other tools miss. For example, configuration analysis [13, 28] cannot find bugs that occur in router software. Second, since data-plane state has relatively simple formats and semantics that are common across many higher-layer protocols and implementations, it simplifies rigorous analysis of a network.

Early data plane verification algorithms were developed in [27], and systems include FlowChecker [9], Anteater [19], and Header Space Analysis [17]. The latter two systems were applied to operational networks and uncovered multiple real-world bugs, validating the data plane analysis approach. However, as noted previously, these are offline rather than real-time systems.

VeriFlow performs real-time data plane verification in the context of software defined networks (SDNs). An SDN comprises, at a high level, (1) a standardized and open interface to read and write the data plane of network devices such as switches and routers; (2) a *controller*, a logically centralized device that can run custom code and is responsible for transmitting commands (forwarding rules) to network devices.

SDNs are a good match for data plane verification. First, a standardized data plane interface such as OpenFlow [6] simplifies unified analysis across all network devices. Second, SDNs ease *real-time* data plane verification since the stream of updates to the network is observable at the controller.

SDN thus simplifies VeriFlow's design. Moreover, we believe SDNs can benefit significantly from VeriFlow's data plane verification layer: the network operator can verify that the network's forwarding behavior is correct, without needing to inspect (or trust) relatively complex controller code, which may be developed by parties outside the network operator's control.

3 Design of VeriFlow

Checking network-wide invariants in the presence of complex forwarding elements can be a hard problem. For example, packet filters alone make reachability checks NP-Complete [19]. Aiming to perform these checks in real-time is therefore challenging. Our design tackles this problem as follows. First, we monitor all the network update events in a live network as they are generated by network control applications, the devices, or the network operator. Second, we confine our verification activities to only those parts of the network whose actions may be influenced by a new update. Third, rather than checking invariants with a general-purpose tool such as a SAT or BDD solver as in [10, 19] (which are generally too slow), we use a custom algorithm. We now discuss each of these design decisions in detail.

VeriFlow's first job is to track every forwarding-state change event. For example, in an SDN such as OpenFlow [21], a centralized controller issues forwarding rules to the network devices to handle flows initiated by users. VeriFlow must intercept all these rules and verify them before they reach the network. To achieve this goal, VeriFlow is implemented as a shim layer between the controller and the network, and monitors all communication in either direction.

For every rule insertion/deletion message, VeriFlow must verify the effect of the rule on the network at very high speed. VeriFlow cannot leverage techniques used by past work [9, 17, 19], because these operate at timescales of seconds to hours. Unlike previous solutions, we do not want to check the entire network on each change. We solve this problem in three steps. First, using the new rule and any overlapping existing rules, we slice the network into a set of *equivalence classes* (ECs) of packets (§ 3.1). Each EC is a set of packets that experience the same forwarding actions throughout the network. Intuitively, each change to the network will typically only affect a very small number of ECs (see § 5.1). Therefore, we find the set of ECs whose operation could be altered by a rule, and verify network invariants only within those classes. Second, VeriFlow builds individual *forwarding graphs* for every modified EC, representing the network's forwarding behavior (§ 3.2). Third, VeriFlow traverses these graphs (or runs custom user-defined code) to determine the status of one or more invariants (§ 3.3). The following subsections describe these steps in detail. Figure 1 shows the placement and operations of VeriFlow in an SDN.

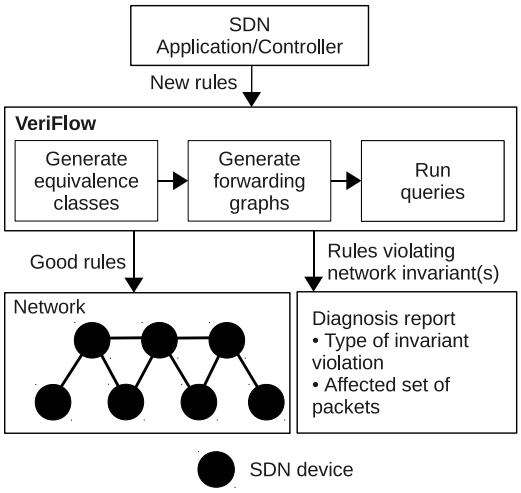


Figure 1: VeriFlow sits between the SDN applications and devices to intercept and check every rule entering the network.

3.1 Slicing the network into equivalence classes

One way to verify network properties is to prepare a model of the entire network using its current data-plane state, and run queries on this model [9, 19]. However, checking the entire network's state every time a new flow rule is inserted is wasteful, and fails to provide real-time response. Instead, we note that most forwarding rule changes affect only a small subset of all possible packets. For example, inserting a longest-prefix-match rule for the destination IP field will only affect forwarding for packets destined to that prefix. In order to confine our verification activities to only the affected set of packets, we slice the network into a set of equivalence classes (ECs) based on the new rule and the existing rules that overlap with the new rule. An equivalence class is defined as follows.

Definition (Equivalence Class): An equivalence class (EC) is a set P of packets such that for any $p_1, p_2 \in P$ and any network device R, the forwarding action is identical for p_1 and p_2 at R.

Separating the entire packet space into individual ECs allows VeriFlow to pinpoint the affected set of packets if a problem is discovered while verifying a newly inserted forwarding rule.

Let us look at an example. Consider an OpenFlow switch with two rules matching packets with destination IP address prefixes 11.1.0.0/16 and 12.1.0.0/16, respectively. If a new rule matching destination IP address prefix 11.0.0.0/8 is added, it may affect packets belonging to the 11.1.0.0/16 range depending on the rules' priority values [6] (the longer prefix may not have higher priority). However, the new rule will not affect packets outside the range 11.0.0.0/8, such as 12.1.0.0/16. Therefore, VeriFlow will only consider the new rule (11.0.0.0/8) and the existing overlapping rule (11.1.0.0/16) while analyzing network properties. These two overlapping rules produce three ECs (represented using the lower and upper bound range values of the destination IP address field): 11.0.0.0 to 11.0.255.255, 11.1.0.0 to 11.1.255.255, and 11.2.255.255 to 11.255.255.255.

VeriFlow needs an efficient data structure to quickly store new network rules, find overlapping rules, and compute the affected ECs. For this we utilize a *multi-dimensional prefix tree (trie)* inspired by traditional packet classification algorithms [26].

A trie is an ordered tree data structure that stores an associative array. In our case, the trie associates the set of packets matched by a forwarding rule with the forwarding rule itself. Each level in the trie corresponds to a specific bit in a forwarding rule (equivalently, a bit in the packet header). Each node in our trie has three branches, corresponding to three possible values that the rule can match: 0, 1, and * (wildcard). The trie can be seen as a composition of several *sub-tries* or dimensions, each corresponding to a packet header field. We maintain a sub-trie in our multi-dimensional trie for each of the mandatory match and packet header fields supported by OpenFlow 1.1.0.[2] (Note that an optimization in our implementation uses a condensed set of fields in the trie;

[2](DL_SRC, DL_DST, NW_SRC, NW_DST, IN_PORT, DL_VLAN, DL_VLAN_PCP, DL_TYPE, NW_TOS, NW_PROTO, TP_SRC, TP_DST, MPLS_LABEL and MPLS_TC).

see § 4.2.) For example, the sub-trie representing the IPv4 destination corresponds to 32 levels in the trie. One of the sub-tries (DL_SRC in our design) appears at the top, the next field's sub-tries are attached to the leaves of the first, and so on (Figure 2). A path from the trie's root to a leaf of one of the bottommost sub-tries thus represents the set of packets that a rule matches. Each leaf stores the rules that match that set of packets, and the devices at which they are located (Figure 2).

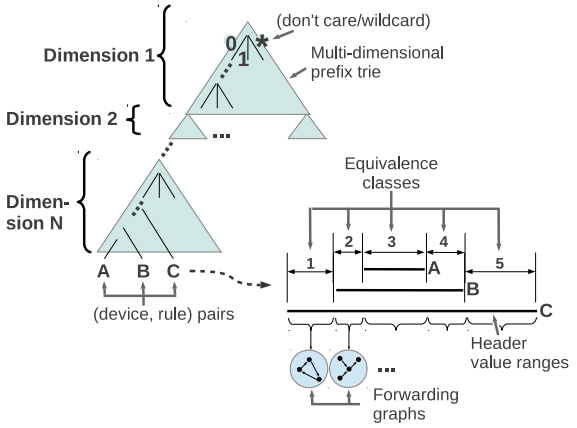


Figure 2: VeriFlow's core algorithmic process.

When a new forwarding rule is generated by the application, we perform a lookup in our trie, by traversing it dimension by dimension to find all the rules that intersect the new rule. At each dimension, we narrow down the search area by only traversing those branches that fall within the range of the new rule using the field value of that particular dimension. The lookup procedure results in the selection of a set of leaves of the bottommost dimension, each with a set of forwarding rules. These rules collectively define a set of packets (in particular, their corresponding forwarding rules) that could be affected by the incoming forwarding rule. This set may span multiple ECs. We next compute the individual ECs as illustrated in Figure 2. For each field, we find a set of disjoint ranges (lower and upper bound) such that no rule splits one of the ranges. An EC is then defined by a particular choice of one of the ranges for each of the fields. This is not necessarily a minimal set of ECs; for example, ECs 2 and 4 in Figure 2 could have been combined into a single EC. However, this method performs well in practice.

3.2 Modeling forwarding state with forwarding graphs

For each EC computed in the previous step, VeriFlow generates a *forwarding graph*. Each such graph is a representation of how packets within an EC will be forwarded through the network. In the graph, a node represents an EC at a particular network device, and a directed edge represents a forwarding decision for a particular (EC, device) pair. Specifically, an edge $X \rightarrow Y$ indicates that according to the forwarding table at node X, packets within this EC are forwarded to Y. To build the graph for each EC, we traverse our trie a second time to find the devices and rules that match packets from that EC. The second traversal is needed to find all those rules that were not necessary to compute the affected ECs in the first traversal, yet can still influence their forwarding behavior. For example, for a new rule with 10.0.0.0/8 specified as the destination prefix, an existing 0.0.0.0/0 rule will not contribute to the generation of the affected ECs, but may influence their forwarding behavior depending on its priority. Given the range values of different fields of an EC, looking up matching rules from the trie structure can be performed very quickly. Here, VeriFlow only has to traverse those branches of the trie having rules that can match packets of that particular EC.

3.3 Running queries

Above, we described how VeriFlow models the behavior of the network using forwarding graphs, building forwarding graphs only for those equivalence classes (ECs) whose behavior may have changed. Next, we answer queries (check invariants) using this model.

VeriFlow maintains a list of invariants to be checked. When ECs have been modified, VeriFlow checks each (invariant, modified EC) pair. An invariant is specified as a verification function that takes as input the forwarding graph for a specific EC, performs arbitrary computation, and can trigger resulting actions. VeriFlow exposes an API (Application Programming Interface), the implementation of which is described in § 4.3, so that new invariants can be written and plugged in.

Up to a certain level of detail, the forwarding graph is an exact representation of the forwarding behavior of the network. Therefore, invariant modules can check a large diversity of conditions concerning network behavior. For example:

- Basic reachability: The verification function traverses the directed edges in the forwarding graph (using depth-first search in our implementation) to determine whether packets will be delivered to the destination address specified in the rule.
- Loop-freeness: The verification function traverses the given EC's forwarding graph to check that it does not contain a loop.
- Consistency: Given two (pre-specified) routers R_1, R_2 that are intended to have identical forwarding operations, the verification function traverses

the forwarding graph starting at R_1 and R_2 to test whether the fate of packets is the same in both cases. (Any difference may indicate a bug.)

Further examples include detecting "black holes" where packets are dropped, ensuring isolation of multiple VLANs, verifying access control policies, checking whether a new rule conflicts with an existing rule, checking whether an EC changes its next hop due to the insertion/deletion of a rule, ensuring that packets always traverse a firewall, and so on.

There are two key limitations on what invariants can be feasibly implemented. First, VeriFlow's forwarding graph construct must include the necessary information. Our current implementation of VeriFlow does not, for example, incorporate information on buffer sizes that would be necessary for certain performance invariants. (There is not, however, any fundamental reason that VeriFlow could not be augmented with such metadata.) Second, the invariant check must be implementable in the *incremental* manner described above where only the modified ECs are considered at each step.

If a verification function finds a violated invariant, it can choose to trigger further actions within VeriFlow. Two obvious actions are dropping the rule that was being inserted into the network, or installing the rule but generating an alarm for the operator. For example, the operator could choose to drop rules that cause a security violation (such as packets leaking onto a protected VLAN), but only generate an alarm for a black hole. Since verification functions are arbitrary code, they may take other actions as well, such as maintaining statistics (e.g., rate of forwarding behavior change) or writing logs.

3.4 Dealing with high verification time

VeriFlow achieves real-time response by confining its verification activities within those parts of the network that are affected when a new forwarding rule is installed. In general, the effectiveness of this approach will be determined by numerous factors, such as the complexity of verification functions, the size of the network, the number of rules in the network, the number of unique ECs covered by a new rule, the number of header fields used to match packets by a new rule, and so on.

However, perhaps the most important factor summarizing verification time is the *number of ECs modified*. As our later experiments will show, VeriFlow's verification time is roughly linear in this number. In other words, VeriFlow has difficulty verifying invariants in real-time when large swaths of the network's forwarding behavior are altered in one operation.

When such disruptive events occur, VeriFlow may need to let new rules be installed in the network without waiting for verification, and run the verification process in parallel. We lose the ability to block problematic rules before they enter the network, but we note several mitigating facts. First, the most prominent example of a disruptive event affecting many ECs is a link failure, in which case VeriFlow anyway cannot block the modification from entering the network. Second, upon (eventually) detecting a problem, VeriFlow can still raise an alarm and remove the problematic rule(s) from the network. Third, the fact that the number of affected ECs is large may itself be worthy of an immediate alarm even before invariants are checked for each EC. Finally, our experiments with realistic forwarding rule update traces (§ 5) show that disruptive events (i.e., events affecting large number of ECs) are rare: in the vast majority of cases (around 99%), the number of affected ECs is small (less than 10).

4 Implementation

We describe three key aspects of our implementation: our shim layer to intercept network events (§ 4.1), an optimization to accelerate verification (§ 4.2), and our API for custom invariants (§ 4.3).

4.1 Making deployment transparent

In order to ease the deployment of VeriFlow in networks with OpenFlow-enabled devices, and to use VeriFlow with unmodified OpenFlow applications, we need a mechanism to make VeriFlow transparent so that these existing OpenFlow entities may remain unaware of the presence of VeriFlow. We built two versions of VeriFlow. One is a *proxy process* [25] that sits between the controller and the network, and is therefore independent of the particular controller. The second version is *integrated* with the NOX OpenFlow controller [14] to improve performance; our performance evaluation is of this version. We expect one could similarly integrate VeriFlow with other controllers, such as Floodlight [2], Beacon [1] and Maestro [11], without significant trouble.

We built our implementation within NOX version 0.9.1 (full beta single-thread version). We integrated VeriFlow within NOX, enabling it to run as a transparent rule verifier sitting between the OpenFlow applications implemented using NOX's API, and the switches and routers in the network. SDN applications running on NOX use the NOX API to manipulate the forwarding state of the network, resulting in OFPT_FLOW_MOD (flow table modification) and other OpenFlow messages generated by NOX. We modified NOX to intercept these messages, and redirect them to our VeriFlow module. This ensures that all messages are intercepted by VeriFlow before they are dispatched to the network. Veri-

Flow then processes and checks the forwarding rules contained in these messages for correctness, and can block problematic flow rules.

To integrate the VeriFlow module, we extend two parts of NOX. First, within the core of NOX, the *send_openflow_command()* interface is responsible for adding (relaying) flow rules from OpenFlow applications to the switches. At the lower layers of NOX, *handle_flow_removed()* handles events that remove rules from switches, due to rule timeouts or commands sent by applications. Our implementation intercepts all messages sent to these two function calls, and redirects them to VeriFlow. To reduce memory usage and improve running time, we pass these messages via shallow copy.

There are five types of flow table modification messages that can be generated by OpenFlow applications: OFPFC_ADD, OFPFC_MODIFY_STRICT, OFPFC_DELETE_STRICT, OFPFC_MODIFY and OFPFC_DELETE. These rules differ in terms of whether they add, modify or delete a rule from the flow table. The strict versions match all the fields bit by bit, whereas the non-strict versions allow wildcards. Our implementation handles all these message types appropriately.

4.2 Optimizing the verification process

We use an optimization technique that exploits the way certain match and packet header fields are handled in the OpenFlow 1.1.0 specification. 10 out of 14 fields in this specification do not support arbitrary wildcards.[3] One can only specify an exact value or the special *ANY* (wildcard) value in these fields. We do not use separate dimensions in our trie to represent these fields, because we do not need to find multiple overlapping ranges for them. Therefore, we only maintain the trie structure for the other four fields (DL_SRC, DL_DST, NW_SRC and NW_DST). Due to this change, we generate the set of affected equivalence classes (ECs) in three steps. First, we use the trie structure to look for network-wide overlapping rules, and find the set of affected packets determined by the four fields that are represented by the trie. Each individual packet set we get from this step is actually a set of ECs that can be distinguished by the other 10 fields. Second, for each of these packet sets, we extract all the rules that can match packets of that particular class from the location/device of the newly inserted rule. We linearly go through all these rules to find non-overlapping range values for the rest of the fields that are not maintained in the trie structure. Thus, each packet set found in the first step breaks into multiple finer packet sets spanning all the 14 mandatory OpenFlow match and packet header fields. Note that in this step we only consider the rules present at the device of the newly inserted rule. Therefore, in the final step, as we traverse the forwarding graphs, we may encounter finer rules at other devices that will generate new packet sets with finer granularity. We handle them by maintaining sets of excluded packets as described in the next paragraph.

Each forwarding graph that we generate using our trie structure represents the forwarding state of a group of packet sets that can be distinguished using the 10 fields that do not support arbitrary wildcards. Therefore, while traversing the forwarding graphs, we only work on those rules that overlap with the newly inserted rule on these 10 fields. As we move from node to node while traversing these graphs, we keep track of the ECs that have been served by finer rules and are no longer present in the primary packet set that was generated in the first place. For example, in a device, a subset of a packet set may be served by a finer rule having higher priority than a coarser rule that serves the rest of that packet set. We handle this by maintaining a set of excluded packets for each forwarding action. Therefore, whenever we reach a node that answers a query (e.g., found a loop or reached a destination), the primary packet set minus the set of excluded packets gives the set of packets that experiences the result of the query.

4.3 API to write general queries

We expose a set of functions that can be used to write general queries in C++. Below is a list of these functions along with the required parameters.

GetAffectedEquivalenceClasses: Given a new rule, this function computes the set of affected ECs, and returns them. It also returns a set of sub-tries from the last dimension of our trie structure. Each sub-trie holds the rules that can match packets belonging to one of the affected ECs. This information can be used to build the forwarding graphs of those ECs. This function takes the following parameters.
- Rule: A newly inserted rule.
- *Returns:* Affected ECs.
- *Returns:* Sub-tries representing the last dimension, and holding rules that can match packets of the affected ECs.

GetForwardingGraph: This function generates and returns the forwarding graph for a particular EC. It takes the following parameters.
- EquivalenceClass: An EC whose forwarding graph will be computed.
- TrieSet: Sub-tries representing the last dimension, and holding rules that match the EC supplied as the first argument.
- *Returns:* Corresponding forwarding graph.

ProcessCurrentHop: This function allows the user to

[3] IN_PORT, DL_VLAN, DL_VLAN_PCP, DL_TYPE, NW_TOS, NW_PROTO, TP_SRC, TP_DST, MPLS_LABEL and MPLS_TC.

traverse a forwarding graph in a custom manner. Given a location and EC, it returns the corresponding next hop. It handles the generation of multiple finer packet sets by computing excluded packet sets that need to be maintained because of our optimization strategy (§ 4.2). Due to this optimization, this function returns a set of (next hop, excluded packet set) tuples — effectively, an annotated directed edge in the forwarding graph. With repeated calls to this function across nodes in the forwarding graphs, custom invariant-checking modules can traverse the forwarding graph and perform arbitrary computation on its structure. This function takes the following parameters.
- ForwardingGraph: The forwarding graph of an EC.
- Location: The current location of the EC.
- *Returns:* (Next hop, excluded packet set) tuples.

Let us look at an example that shows how this API can be used in practice. A network operator may want to ensure that packets belonging to a certain set always pass through a firewall device. This invariant can be violated during addition/deletion of rules, or during link up/down events. To check this invariant, the network operator can extend VeriFlow using the above API to incorporate a custom query algorithm that generates an alarm when the packet set under scrutiny bypasses the firewall device. In fact, the network operator can implement any query that can be answered using the information present in the forwarding graphs.

5 Evaluation

In this section, we present a performance evaluation of our VeriFlow implementation. As VeriFlow intercepts every rule insertion message whenever it is issued by an SDN controller, it is crucial to complete the verification process in real time so that network performance is not affected, and to ensure scalability of the controller. We evaluated the overhead of VeriFlow's operations with the help of two experiments. In the first experiment (§ 5.1), our goal is to microbenchmark different phases of VeriFlow's operations and observe their contribution to the overall running time. The goal of the second experiment (§ 5.2) is to assess the impact of VeriFlow on TCP connection setup latency and throughput as perceived by end users of an SDN.

In all of our experiments, we used our basic reachability algorithms to test for loops and black holes for every flow modification message that was sent to the network. All of our experiments were performed on a Dell Optiplex 9010 machine with an Intel Core i7 3770 CPU with 4 physical cores and 8 threads at 3.4 GHz, and 32 GB of RAM, running 64 bit Ubuntu Linux 11.10.

5.1 Per-update processing time

In this experiment, we simulated a network consisting of 172 routers following a Rocketfuel [7] topology (AS 1755), and replayed BGP (Border Gateway Protocol) RIB (Routing Information Base) and update traces collected from the Route Views Project [8]. We built an OSPF (Open Shortest Path First) simulator to compute the IGP (Interior Gateway Protocol) path cost between every pair of routers in the network. A BGP RIB snapshot consisting of 5 million entries was used to initialize the routers' FIB (Forwarding Information Base) tables. Only the FIBs of the border routers were initialized in this phase. We randomly mapped Route Views peers to border routers in our network, and then replayed RIB and update traces so that they originate according to this mapping. We replayed a BGP update trace containing 90,000 updates to trigger dynamic changes in the network. Upon receiving an update from the neighboring AS, each border router sends the update to all the other routers in the network. Using standard BGP polices, each router updates its RIB using the information present in the update, and updates its FIB based on BGP AS path length and IGP path cost. We fed all the FIB changes into VeriFlow to measure the time VeriFlow takes to complete its individual steps described in § 3. We recorded the run time to process each change individually. Note that in this first set of experiments, only the destination IP address is used to forward packets. Therefore, only this one field contributes to the generation of equivalence classes (ECs). We initialize the other fields with ANY (wildcards).

The results from this experiment are shown in Figure 3(a). VeriFlow is able to verify most of the updates within 1 millisecond (ms), with mean verification time of 0.38ms. Moreover, of this time, the query phase takes only 0.01ms on an average, demonstrating the value of reducing the query problem to a simple graph traversal for each EC. Therefore, VeriFlow would be able to run multiple queries of interest to the network operator (e.g., black hole detection, isolation of multiple VLANs, etc.) within a millisecond time budget.

We found that the number of ECs that are affected by a new rule strongly influences verification time. The scatter plot of Figure 3(b) shows one data point for each observed number of modified ECs (showing the mean verification time across all rules, which modified that number of ECs). The largest number of ECs affected by a single rule was 574; the largest verification latency was 159.2ms due to an update affecting 511 ECs. However, in this experiment, we found that for most updates the number of affected ECs is small. 94.5% of the updates only affected a single EC, and 99.1% affected less than 10 ECs. Therefore, only a small fraction of rules (0.9%) affected large numbers of ECs. This can be observed by

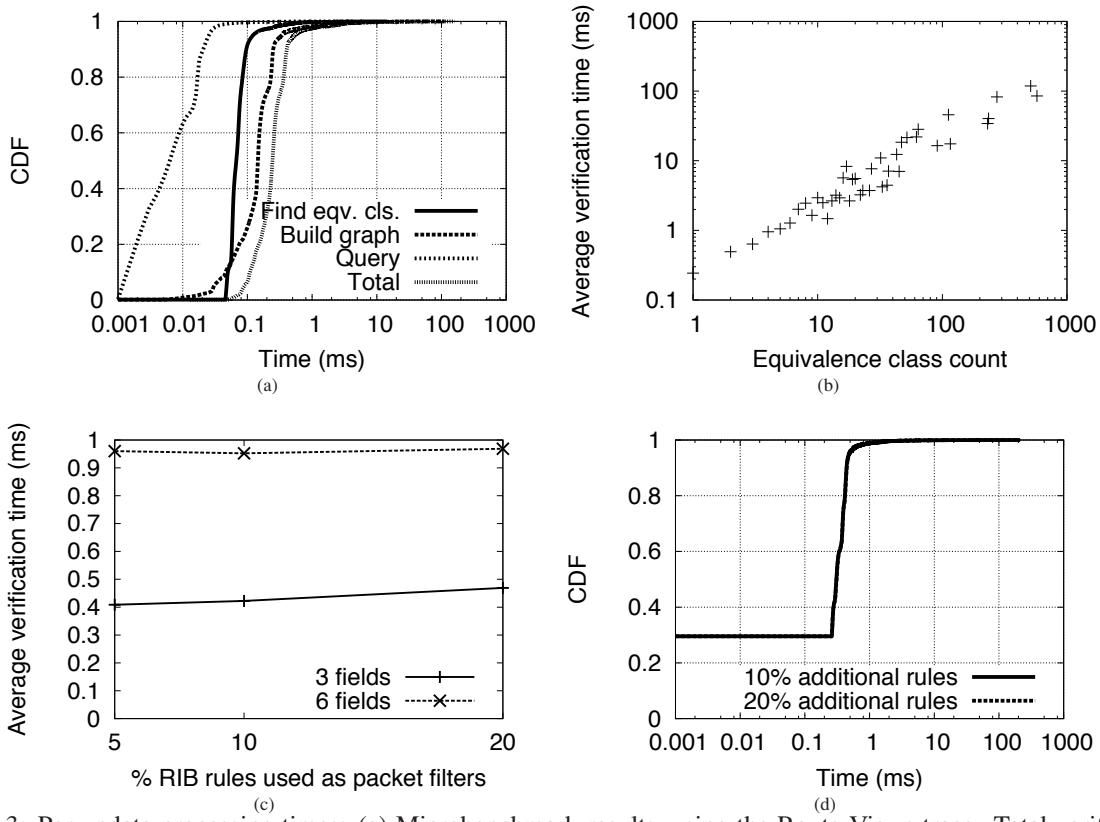


Figure 3: Per-update processing times: (a) Microbenchmark results, using the Route Views trace. Total verification time of VeriFlow remained below 1ms for 97.8% of the updates. (b) Scatter plot showing the influence of number of equivalence classes on verification time. (c) Results from multi-field packet filter experiment using the Route Views trace. As more fields are used in forwarding rules, the running time of VeriFlow increases. The average verification latency is not significantly influenced as we increase the number of filters present in the network. (d) Results from the conflict detection test. VeriFlow is fast enough to compute all the conflicting rules within hundreds of microseconds for 99% of the updates.

looking at the long tail of Figure 3(a).

In the above experiment, we assumed that the network topology remains unchanged, i.e., there are no link or node failures. In case of a link failure or node failure (which can be thought of as failure of multiple links connected to the failed node), the packets that were using that link or node will experience changes in their forwarding behavior. When this happens, VeriFlow's job is to verify the fate of those affected packets. In order to evaluate VeriFlow's performance in this scenario, we used the above topology and traces to run a new experiment. In this experiment, we fed both the BGP RIB trace and update trace to the network. Then we removed each of the packet-carrying links (381 in total) of the network one by one (restoring a removed link before removing the next), and computed the number of affected ECs and the running time of VeriFlow to verify the behavior of those classes. We found that most of the link removals affected a large number of ECs. 254 out of 381 links affected more that 1,000 ECs. The mean verification time to verify a link failure event was 1.15 seconds, with a maximum of 4.05 seconds. We can deal with such cases by processing the forwarding graphs of different ECs in parallel on multi-core processors. This is possible because the forwarding graphs do not depend on each other, or on any shared data structure. However, as link or node failures cannot be avoided once they happen, this may not be a serious issue for network operators.

In order to evaluate VeriFlow's performance in the presence of more fields, we changed the input data set to add packet filters that will selectively drop packets after matching them against multiple fields. We randomly selected a subset of the existing RIB rules currently present in the network, and inserted packet filter rules by specifying values in some of the other fields that were not present in the original trace. We ran this experiment with two sets of fields. In the first set we used TP_SRC and TP_DST in addition to NW_DST (3 fields in total), which was already present in the trace. For each randomly selected RIB rule, we set random values to those

two fields (TP_SRC and TP_DST), and set its priority higher than the original rule. The remaining 11 fields are set to ANY. While replaying the updates, all the 14 fields except NW_DST are set to ANY.

In the second set we used NW_SRC, IN_PORT, DL_VLAN, TP_SRC and TP_DST in addition to NW_DST (6 fields in total). For each randomly selected RIB rule, we set random values to IN_PORT, DL_VLAN, TP_SRC and TP_DST, a random /16 value in NW_SRC, and set the priority higher than the original rule. The remaining 8 fields are set to ANY. While replaying the updates, all the 14 fields except NW_SRC and NW_DST are set to ANY. In the updates, the NW_SRC is set to a random /12 value and the NW_DST is the original value present in the trace. We ran this experiment multiple times varying the percentage of RIB rules that are used to generate random filter rules with higher priority.

Figure 3(c) shows the results of this experiment. Verification time is heavily affected by the number of fields used to classify packets. This happens because as we use more fields to classify packets at finer granularities, more unique ECs are generated, and hence more forwarding graphs need to be verified. We also note from Figure 3(c) that VeriFlow's overall performance is not affected much by the number of filters that we install into the network.

In all our experiments thus far, we kept a fixed order of packet header fields in our trie structure. We started with DL_SRC (DS), followed by DL_DST (DD), NW_SRC (NS) and NW_DST (ND). In order to evaluate the performance of VeriFlow with different field orderings, we re-ran the above packet filter experiment with reordered fields. In all the runs we used random values for the NW_SRC field and used the NW_DST values present in the Route Views traces. All the other fields were set to ANY. We installed random packet filter rules for 10% of the BGP RIB entries. As our dataset only specified values for the NW_SRC and NW_DST fields, there were a total of 12 different orderings of the aforementioned 4 fields. Table 1 shows the results from this experiment.

Table 1: Effect of different field orderings on total running time of VeriFlow.

Order	Time (ms)	Order	Time (ms)
DS-DD-NS-ND	1.001	DS-DD-ND-NS	0.090
DS-NS-DD-ND	1.057	DS-ND-DD-NS	0.096
NS-DS-DD-ND	1.144	ND-DS-DD-NS	0.101
NS-DS-ND-DD	1.213	ND-DS-NS-DD	0.103
NS-ND-DS-DD	1.254	ND-NS-DS-DD	0.15
DS-NS-ND-DD	1.116	DS-ND-NS-DD	0.098

From Table 1, we can see that changing the field order in the trie structure greatly influences the running time of VeriFlow. Putting the NW_DST field ahead of NW_SRC reduced the running time by an order of magnitude (from around 1ms to around 0.1ms). This happens because a particular field order may produce fewer unique ECs compared to other field orderings for the same rule. However, it is difficult to come up with a single field order that works best in all scenarios, because it is highly dependent on the type of rules present in a particular network. Changing the field order in the trie structure dynamically and efficiently as the network state evolves would be an interesting area for future work.

Checking non-reachability invariants: Most of our discussion thus far focused on checking invariants associated with the inter-reachability of network devices. To evaluate the generality of our tool, we implemented two more invariants using our API that were not directly related to reachability: *conflict detection* (whether the newly inserted rule violates isolation of flow tables between network slices, accomplished by checking the output of the EC search phase), and *k-monitoring* (ensuring that all paths in the network traverse one of several deployed monitoring points, done by augmenting the forwarding graph traversal process). We found that the overhead of these checks was minimal. For the conflict detection query, we ran the above filtering experiment using the 6-field set with 10% and 20% newly inserted random rules. However, this time instead of checking the reachability of the affected ECs as each update is replayed, we only computed the set of rules that overlap/conflict with the newly inserted rule. The results from this experiment are shown in Figure 3(d).

From this figure, we can see that conflicting rule checking can be done quickly, taking only 0.305ms on average. (The step in the CDF is due to the fact that some withdrawal rules did not overlap with any existing rule.)

For the k-monitoring query experiment, we used a snapshot of the Stanford backbone network data-plane state that was used in [17]. This network consists of 16 routers, where 14 of these are internal routers and the other 2 are gateway routers used to access the outside network. The snapshot contains 7,213 FIB table entries in total. In this experiment, we used VeriFlow to test whether *all* the ECs currently present in the network pass through one of the two gateway routers of the network. We observed that at each location the average latency to perform this check for all the ECs is around 68.06ms with a maximum of 75.39ms.

5.2 Effect on network performance

In order to evaluate the effect of VeriFlow's operations on user-perceived TCP connection setup latency and the network throughput, we emulated an OpenFlow network consisting of 172 switches following the aforementioned Rocketfuel topology using Mininet [3]. Mininet creates

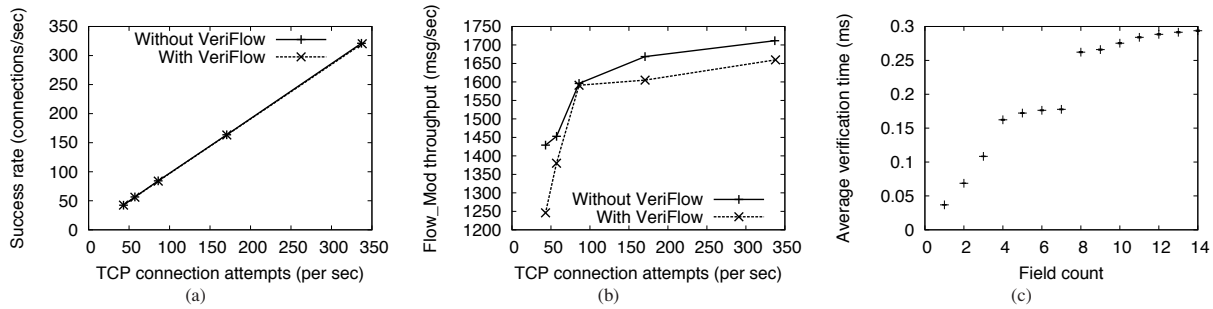


Figure 4: Effect on network performance: (a) TCP connection setup throughput, and (b) Throughput of flow modification (Flow_Mod) messages, with and without VeriFlow. For different loads, VeriFlow imposes minimal overhead. (c) Effect of the number of packet header fields on VeriFlow's verification speed. As we increase the number of fields, overhead of VeriFlow increases gradually.

a software-defined network (SDN) with multiple nodes on a single machine. We connected one host to every switch in this emulated network. We ran the NOX OpenFlow controller along with an application that provides the functionality of a learning switch. It allows a host to reach any other host in the network by installing flow rules in the switches using flow modification (Flow_Mod) messages. We implemented a simple TCP server program and a simple TCP client program to drive the experiment. The server program accepts TCP connections from clients and closes the connection immediately. The client program consists of two threads. The primary thread continuously sends connect requests to a random server using a non-blocking socket. To vary the intensity of the workload, our TCP client program generates connections periodically with a parameterized sleep interval (S). The primary thread at each client sleeps for a random interval between 0 to S seconds (at microsecond granularity) before initiating the connection request, and iterating. The secondary thread at each client uses the *select* function to look for connections that are ready for transmission or experienced an error. A user supplied polling interval (P) is used to control the rate at which the select call will return. We set P inversely proportional to the S value to avoid busy waiting and to allow the other processes (e.g., Open vSwitch [5]) to get a good share of the CPU. We ran the server program at each of the 172 hosts, and configured the client programs at all the hosts to continually connect to the server of random hosts (excluding itself) over a particular duration (at least 10 minutes). In the switch application, we set the rule eviction idle timeout to 1 second and hard timeout to 5 seconds.

We ran this experiment first with NOX alone, and then with NOX and VeriFlow. We used the same seed in all the random number generators to ensure similar loads in both the runs. We also varied the S value to monitor the performance of VeriFlow under a range of network loads.

Figure 4(a) shows the number of TCP connections that were successfully completed per second for different workloads both with and without VeriFlow. From this figure, we can see that in all the cases VeriFlow imposes negligible overhead on the TCP connection setup throughput in our emulated OpenFlow network. The largest reduction in throughput that we observed in our experiments was only 0.74%.

Figure 4(b) shows the number of flow modification (Flow_Mod) messages that were processed and sent to the network per second for different workloads both with and without VeriFlow. From this figure, again we can see that in all the cases VeriFlow imposes minimal overhead on the flow modification message throughput. The largest reduction in throughput that we observed in our experiments was only 12.8%. This reduction in throughput is caused by the additional processing time required to verify the flow modification messages before they are sent to the network.

In order to assess the impact of VeriFlow on end-to-end TCP connection setup latency, we ran this experiment with S set to 30 seconds. We found that in the presence of VeriFlow, the average TCP connection setup latency increases by 15.5% (45.58ms without VeriFlow versus 52.63ms with VeriFlow). As setting up a TCP connection between two hosts in our emulated 172 host OpenFlow network requires installing flow rules into more than one switch, the verification performed by VeriFlow after receiving each flow rule from the controller inflates the end-to-end connection setup latency to some extent.

Lastly, we ran this experiment after modifying VeriFlow to work with different numbers of OpenFlow packet header fields. Clearly, if we restrict the number of fields during the verification process, there will be less work for VeriFlow, resulting in faster verification time. In this experiment, we gradually increased the number of OpenFlow packet header fields that were used during the verification process (from 1 to 14). VeriFlow simply ignored the excluded fields, and it reduced the number of dimensions in our trie structure. We set S to 10

seconds and ran each run for 10 minutes. During the runs, we measured the verification latency experienced by each flow modification message generated by NOX, and computed their average at each run.

Figure 4(c) shows the results from this experiment. Here, we see that with the increase in the number of packet header fields, the verification overhead of VeriFlow increases gradually but always remains low enough to ensure real-time response. The 5 fields that contributed most in the verification overhead are DL_SRC, DL_DST, NW_SRC, NW_DST and DL_TYPE. This happened because these 5 fields had different values at different flow rules, and contributed most in the generation of multiple ECs. The other fields were mostly wildcards, and did not generate additional ECs.

Comparison with related work: Finally, we compared performance of our technique with two pieces of related work: the Hassel tool presented in [17] (provided to us by the authors), and a BDD-based analysis tool that we implemented from scratch following the strategy presented in [10] (the original code was not available to us). The authors of [17] provided two copies of their tool, one in Python and one in C, and we evaluated using the better-performing C version. While we note these works solve different problems from our work (e.g., HSA performs static verification, and does it between port pairs), we present these results to put VeriFlow's performance in context. First, we ran Hassel over the snapshot of the Stanford backbone network dataplane state that was used in [17]. We found that Hassel's average time to check reachability between a pair of ports (effectively exploring all ECs for that source-destination pair) was 578.62ms, with a maximum of 6.24 seconds. In comparison, VeriFlow took only 68.06ms on average (with a maximum of 75.39ms) to test the reachability of all the ECs currently present at a single node in the network. Next, in the BDD-based approach, we used the NuSMV [4] model checker to build a BDD using a new rule and the overlapping existing rules, and used CTL (Computation Tree Logic) to run reachability queries [10]. Here, we used the Rocketfuel topology and Route Views traces that we used in our earlier experiments. We found that this approach is quite slow and does not provide real-time response while inserting and checking new forwarding rules. Checking an update took 335.71ms on an average with a maximum of 67.16 seconds.

6 Discussion and Future Work

Deciding when to check: VeriFlow may not know when an invariant violation is a true problem rather than an intermediate state during which the violation is considered acceptable by the operator. For example, in an SDN, applications can install rules into a set of switches to build an end-to-end path from a source host to a destination host. However, as VeriFlow is unaware of application semantics, it may not be able to determine these rule set boundaries. This may cause VeriFlow to report the presence of temporary black holes while processing a set of rules one by one. One possible solution is for the SDN application to tell VeriFlow when to check. Moreover, VeriFlow may be used with consistent update mechanisms [20,24], where there are well-defined stages during which the network state is consistent and can be checked.

Handling packet transformations: We can extend our design to handle rules that perform packet transformation such as Network Address Translation. A transformation rule has two parts – the match part determines the set of packets that will undergo the transformation, and the transformation part represents the set of packets into which the matched packets will get transformed. We can handle this case by generating additional equivalence classes and their corresponding forwarding graphs, to address the changes in packet header due to the transformations. In the worst case, if we have transformations at every hop (e.g., in an MPLS network), then we may need to traverse our trie structure multiple times to build an end-to-end path of a particular packet set. We leave a full design and implementation to future work.

Multiple controllers: VeriFlow assumes it has a complete view of the network to be checked. In a multi-controller scenario, obtaining this view in real time would be difficult. Checking network-wide invariants in real time with multiple controllers is a challenging problem for the future.

7 Related Work

Recent work on debugging general networks and SDNs focuses on detecting network anomalies [10, 19], checking OpenFlow applications [12], ensuring data-plane consistency [20, 24], and allowing multiple applications to run side-by-side in a non-conflicting manner [22, 23, 25]. However, unlike VeriFlow, none of the existing solutions provides real-time verification of network-wide invariants as the network experiences dynamic changes.

Checking OpenFlow applications: Several tools have been proposed to find bugs in OpenFlow applications and to allow multiple applications run on the same physical network in a non-conflicting manner. NICE [12] performs symbolic execution of OpenFlow applications and applies model checking to explore the state space of an entire OpenFlow network. Unlike VeriFlow, NICE is a proactive approach that tries to figure out invalid system

states by using a simplified OpenFlow switch model. It is not designed to check network properties in real time. FlowVisor [25] allows multiple OpenFlow applications to run side-by-side on the same physical infrastructure without affecting each others' actions or performance. Unlike VeriFlow, FlowVisor does not verify the rules that applications send to the switches, and does not look for violations of key network invariants.

In [22], the authors presented two algorithms to detect conflicting rules in a virtualized OpenFlow network. In another work [23], Porras et al. extended the NOX OpenFlow controller with a live rule conflict detection engine called FortNOX. Unlike VeriFlow, both of these works only detect conflicting rules, and do not verify the forwarding behavior of the affected packets. Therefore, VeriFlow is capable of providing more useful information compared to these previous works.

Ensuring data-plane consistency: Static analysis techniques using data-plane information suffer from the challenge of working on a consistent view of the network's forwarding state. Although this issue is less severe in SDNs due to their centralized controlling mechanism, inconsistencies in data-plane information may cause transient faults in the network that go undetected during the analysis phase. Reitblatt et al. [24] proposed a technique that uses an idea similar to the one proposed in [15]. By tagging each rule by a version number, this technique ensures that switches forward packets using a consistent view of the network. This same problem has been addressed in [20] using a different approach. While these works aim to tackle transient inconsistencies in an SDN, VeriFlow tries to detect both transient and long-term anomalies as the network state evolves. Therefore, using these above mechanisms along with VeriFlow will ensure that whenever VeriFlow allows a set of rules to reach the switches, they will forward packets without any transient and long-term anomalies.

Checking network invariants: The router configuration checker (rcc) [13] checks configuration files to detect faults that may cause undesired behavior in the network. However, rcc cannot detect faults that only manifest themselves in the data plane (e.g., bugs in router software and inconsistencies between the control plane and the data plane; see [19] for examples).

Anteater [19] uses data-plane information of a network, and checks for violations of key network invariants (absence of routing loops and black holes). Anteater converts the data-plane information into boolean expressions, translates network invariants into instances of boolean satisfiability (SAT) problems, and checks the resultant SAT formulas using a SAT solver. Although Anteater can detect violations of network invariants, it is static in nature, and does not scale well to dynamic changes in the network (taking up to hundreds of seconds to check a single invariant). Header Space Analysis [17] is a system with goals similar to Anteater, and is also not real time.

Concurrent with our work, NetPlumber [16] is a tool based on Header Space Analysis (HSA) that is capable of checking network policies in real time. NetPlumber uses HSA in an incremental manner to ensure real-time response. Unlike VeriFlow, which allows users to write their own custom query procedures, NetPlumber provides a policy language for network operators to specify network policies that need to be checked.

ConfigChecker [10] and FlowChecker [9] convert network rules (configuration and forwarding rules respectively) into boolean expressions in order to check network invariants. They use Binary Decision Diagram (BDD) to model the network state, and run queries using Computation Tree Logic (CTL). VeriFlow uses graph search techniques to verify network-wide invariants, and handles dynamic changes in real time. Moreover, unlike previous solutions, VeriFlow can *prevent* problems from hitting the forwarding plane, whereas FlowChecker find problems after they occur and (potentially) cause damage. ConfigChecker, like rcc, cannot detect problems that only affect the data plane.

An early version of VeriFlow was presented in [18] but only supported checking for a single header field with a basic reachability invariant, had comparatively high overhead, and had a limited evaluation.

8 Conclusion

In this paper, we presented VeriFlow, a network debugging tool to find faulty rules issued by SDN applications, and optionally prevent them from reaching the network and causing anomalous network behavior. VeriFlow leverages a set of efficient algorithms to check rule modification events in real time before they are sent to the live network. To the best of our knowledge, VeriFlow is the first tool that can verify network-wide invariants in a live network in real time. With the help of experiments using a real world network topology, real world traces, and an emulated OpenFlow network, we found that VeriFlow is capable of processing forwarding table updates in real time.

Acknowledgments

We thank our shepherd, Richard Mortier, and the anonymous reviewers for their valuable comments. We gratefully acknowledge the support of the NSA Illinois Science of Security Lablet, and National Science Foundation grants CNS 1040396 and CNS 1053781.

References

[1] Beacon OpenFlow Controller. https://openflow.stanford.edu/display/Beacon/Home.

[2] Floodlight Open SDN Controller. http://floodlight.openflowhub.org.

[3] Mininet: Rapid prototyping for software defined networks. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet.

[4] NuSMV: A new symbolic model checker. http://nusmv.fbk.eu.

[5] Open vSwitch. http://openvswitch.org.

[6] OpenFlow switch specification. http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.

[7] Rocketfuel: An ISP topology mapping engine. http://www.cs.washington.edu/research/networking/rocketfuel.

[8] University of Oregon Route Views Project. http://www.routeviews.org.

[9] AL-SHAER, E., AND AL-HAJ, S. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *SafeConfig* (2010).

[10] AL-SHAER, E., MARRERO, W., EL-ATAWY, A., AND ELBADAWI, K. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *ICNP* (2009).

[11] CAI, Z., COX, A. L., AND NG, T. S. E. Maestro: A system for scalable openflow control. http://www.cs.rice.edu/~eugeneng/papers/TR10-11.pdf.

[12] CANINI, M., VENZANO, D., PERESINI, P., KOSTIC, D., AND REXFORD, J. A NICE way to test OpenFlow applications. In *NSDI* (2012).

[13] FEAMSTER, N., AND BALAKRISHNAN, H. Detecting BGP configuration faults with static analysis. In *NSDI* (2005).

[14] GUDE, N., KOPONEN, T., PETTIT, J., PFAFF, B., CASADO, M., MCKEOWN, N., AND SHENKER, S. NOX: Towards an operating system for networks. In *SIGCOMM CCR* (2008).

[15] JOHN, J. P., KATZ-BASSETT, E., KRISHNAMURTHY, A., ANDERSON, T., AND VENKATARAMANI, A. Consensus routing: The Internet as a distributed system. In *NSDI* (2008).

[16] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *NSDI* (2013).

[17] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *NSDI* (2012).

[18] KHURSHID, A., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying network-wide invariants in real time. In *HotSDN* (2012).

[19] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P. B., AND KING, S. T. Debugging the data plane with Anteater. In *SIGCOMM* (2011).

[20] MCGEER, R. A safe, efficient update protocol for OpenFlow networks. In *HotSDN* (2012).

[21] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., AND SHENKER, S. OpenFlow: Enabling innovation in campus networks. In *SIGCOMM CCR* (2008).

[22] NATARAJAN, S., HUANG, X., AND WOLF, T. Efficient conflict detection in flow-based virtualized networks. In *ICNC* (2012).

[23] PORRAS, P., SHIN, S., YEGNESWARAN, V., FONG, M., TYSON, M., AND GU, G. A security enforcement kernel for OpenFlow networks. In *HotSDN* (2012).

[24] REITBLATT, M., FOSTER, N., REXFORD, J., SCHLESINGER, C., AND WALKER, D. Abstractions for network update. In *SIGCOMM* (2012).

[25] SHERWOOD, R., GIBB, G., YAP, K.-K., APPENZELLER, G., CASADO, M., MCKEOWN, N., AND PARULKAR, G. Can the production network be the testbed? In *OSDI* (2010).

[26] VARGHESE, G. Network Algorithmics: An interdisciplinary approach to designing fast networked devices, 2004.

[27] XIE, G., ZHAN, J., MALTZ, D., ZHANG, H., GREENBERG, A., HJALMTYSSON, G., AND REXFORD, J. On static reachability analysis of IP networks. In *INFOCOM* (2005).

[28] YUAN, L., MAI, J., SU, Z., CHEN, H., CHUAH, C.-N., AND MOHAPATRA, P. FIREMAN: A toolkit for firewall modeling and analysis. In *SnP* (2006).

Software Defined Traffic Measurement with OpenSketch

Minlan Yu[†] *Lavanya Jose*[*] *Rui Miao*[†]
[†] *University of Southern California* [*] *Princeton University*

Abstract

Most network management tasks in software-defined networks (SDN) involve two stages: measurement and control. While many efforts have been focused on network control APIs for SDN, little attention goes into measurement. The key challenge of designing a new measurement API is to strike a careful balance between generality (supporting a wide variety of measurement tasks) and efficiency (enabling high link speed and low cost). We propose a software defined traffic measurement architecture OpenSketch, which separates the measurement data plane from the control plane. In the data plane, OpenSketch provides a simple three-stage pipeline (hashing, filtering, and counting), which can be implemented with commodity switch components and support many measurement tasks. In the control plane, OpenSketch provides a measurement library that automatically configures the pipeline and allocates resources for different measurement tasks. Our evaluations of real-world packet traces, our prototype on NetFPGA, and the implementation of *five* measurement tasks on top of OpenSketch, demonstrate that OpenSketch is general, efficient and easily programmable.

1 Introduction

Recent advances in software-defined networking (SDN) have significantly improved network management. Network management involves two important stages: (1) measuring the network in real time (e.g., identifying traffic anomalies or large traffic aggregates) and then (2) adjusting the control of the network accordingly (e.g., routing, access control, and rate limiting). While there have been many efforts on designing the right APIs for network *control* (e.g., OpenFlow [29], ForCES [1], rule-based forwarding [33], etc.), little thought has gone into designing the right APIs for *measurement*. Since control and measurement are two important halves of network management, it is important to design and build a new software-defined measurement architecture. The key challenge is to strike a careful balance between generality (supporting a wide variety of measurement tasks) and efficiency (enabling high link speed and low cost).

Flow-based measurements such as NetFlow [2] and sFlow [42] provide generic support for different measurement tasks, but consume too resources (e.g., CPU, memory, bandwidth) [28, 18, 19]. For example, to identify the big flows whose byte volumes are above a threshold (i.e., heavy hitter detection which is important for traffic engineering in data centers [6]), NetFlow collects flow-level counts for *sampled* packets in the data plane. A high sampling rate would lead to too many counters, while a lower sampling rate may miss flows. While there are many NetFlow improvements for specific measurement tasks (e.g., [48, 19]), a different measurement task may need to focus on small flows (e.g., anomaly detection) and thus requiring another way of changing NetFlow. Instead, *we should provide more customized and dynamic measurement data collection defined by the software written by operators based on the measurement requirements; and provide guarantees on the measurement accuracy.*

As an alternative, many *sketch*-based streaming algorithms have been proposed in the theoretical research community [7, 12, 46, 8, 20, 47], which provide efficient measurement support for individual management tasks. However, these algorithms are not deployed in practice because of their lack of generality: Each of these algorithms answers just one question or produces just one statistic (e.g., the unique number of destinations), so it is too expensive for vendors to build new hardware to support each function. For example, the Space-Saving heavy hitter detection algorithm [8] maintains a hash table of items and counts, and requires customized operations such as keeping a pointer to the item with minimum counts and replacing the minimum-count entry with a new item, if the item does not have an entry. Such al-

gorithms require not only a customized switch chip (or network processor) to implement, but also are hard to change for a better solution in the future. Instead, we should *design a simple, efficient data plane that is easy to implement with commodity switch components, while leaving those customized data analysis to the software part in the controller.*

Inspired by OpenFlow which enables simple and efficient control of switches by separating control and data functions, we design and implement a new *software-defined traffic measurement* architecture *OpenSketch*. OpenSketch provides a generic and efficient measurement solution, by separating the measurement control and data plane functions (Figure 1). Like OpenFlow, OpenSketch keeps the data plane simple to implement and flexible to configure, while enabling flexible and easy programming for different measurement tasks at the controller. OpenSketch's measurement support, together with OpenFlow-like control support, can form a complete measure and control loop for software-defined networking. We expect that OpenSketch will foster more network management solutions using measurement data and theoretical innovations in measurement algorithms.

We made two major contributions in OpenSketch:

First, OpenSketch redesigns the measurement APIs at switches to be both *generic* and *efficient*. Unlike flow-based measurement, OpenSketch allows more customized and thus more efficient data collection with respect to choosing which flow to measure (using both hashing and wildcard rules), which data to measure (more than just byte/packet counters, such as the average flow size), and how to store the measurement data (more compact data structures rather than simple five tuples plus per-flow counters). We design a three-stage data plane pipeline that supports many measurement tasks with simple configurations, operates at high link speed with limited memory, and works with commodity hardware components.

Second, OpenSketch makes measurement programming easier at the controllers by freeing operators from understanding the complex switch implementations and parameter tuning in diverse sketches. We build a measurement library which automatically configures the data plane pipeline for different sketches and allocates the switch memory across tasks to maximize accuracy. The OpenSketch measurement library also makes it easier for operators to apply new theoretical research results of sketches and streaming algorithms upon commodity switch components.

We compare OpenSketch with NetFlow and streaming algorithms using packet traces from CAIDA [41], and show that OpenSketch provides a good accuracy-memory tradeoff. We build an OpenSketch data plane prototype on NetFPGA, which shows no additional overhead on switch data plane. Both OpenSketch data and control plane codes are publicly available [5].

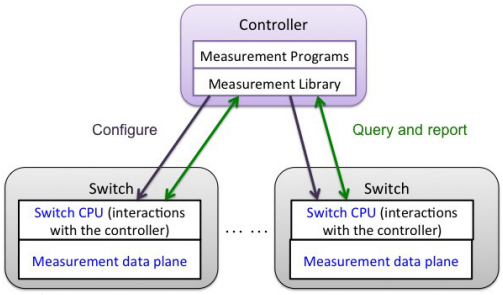


Figure 1: Software defined traffic measurement

Although OpenSketch is sketch-based, it can support a wide variety of measurement tasks because different sketches already support tasks including counting a set of flows, measuring various traffic statistics, and identifying specific flows. In order to make OpenSketch simple enough to implement with commodity hardware and operate at line rate, OpenSketch does not support all the traffic measurement tasks. For example, the OpenSketch data plane does not provide complex data structures (e.g, binary search tree used in some sketches [9]) or directly support all measurement algorithms (e.g., flow size distribution). Instead, we rely on the software in the controller to implement these complex data structures and algorithms using simpler sketches in the data plane.

With our OpenSketch platform, we implement *five* measurement tasks (e.g., heavy hitter detection, flow size distribution calculation) using the measurement library with 15-170 lines of code.

2 Background on Sketches

Sketches are compact data structures used in streaming algorithms to store summary information about the state of packets. Compared to flow-based counters, sketches have two key properties:

(1) Low memory usage: The size of summary information (sketch outputs) is significantly smaller than the input size. For example, the bitmap [21] is a simple sketch that maintains an array of bits to count the number of unique elements (e.g., IP source addresses). We hash each item in a stream of elements to one of the b bits in the bitmap and set the bit to 1. The number of unique elements can be estimated by $b \times \ln(b/z)$ where z is the number of unset bits. Another example is the Count-Min sketch, which maintains a two dimensional array A of counters with width w and depth k. Each entry in the array is initially zero. For each element x, we perform the k pair-wise independent hash functions and increment the counts at $A[i,h_i(x)](i = 1..k)$. To get the frequency for an element x, we just perform the same k hash functions,

get k counters from the Count-Min sketch, and report the minimum counter among them.

(2) Provable tradeoffs of memory and accuracy: Sketches often provide a provable tradeoff between memory and accuracy, though the definition of accuracy depends on the actual sketch function. For a bitmap of b bits, the error in the estimated count $\hat{n}$ compared to the real value n is $SD(\hat{n}/n) \approx \sqrt{e^{\rho} - \rho - 1}/(\rho\sqrt{b})$, where ρ is the average number of flows that hash to a bit [21]. In the Count-Min sketch, the relative error in terms of total count is $\varepsilon_{cm} = e \times t \times H_{cm}/C_{cm}$, where H_{cm} is the number of hash functions and the e is Euler's constant, t is the number of bytes per counter, and C_{cm} is its total memory in bytes. Note that the bound is for the worst-case traffic and thus independent of the traffic characteristics. If we have a better understanding of the traffic then we can have a tighter bound for the memory-accuracy tradeoff. For example, for the Count-Min sketch, if we can estimate the distribution skew of the elements in the stream (e.g., a Zipfian parameter of α), we can have a tighter estimation of the error rate $(e \times t \times H_{cm}/C_{cm})^{\max(1,\alpha)}$ [14].

Example measurement solutions built on sketches: Sketches can be used for many measurement tasks such as heavy hitters detection [8], traffic change detection [36], flow size distribution estimation [27], global iceberg detection [25], and fine-grained delay measurement [35]. For example, to count the number of unique sources accessing a web server via port 80, we can simply filter the traffic based on port 80 and then use a bitmap to count the number of unique source addresses. Another example is heavy hitter detection [8], which is important for traffic engineering in data centers [6]. Heavy hitters are those large flows that consume more than a fraction T of the link capacity during a time interval. To identify heavy hitters, we first use a Count-Min sketch to maintain the counts for each flow. Then, we identify potential flows that hashed to heavy counters in a reversible sketch [36], and verify their actual count using the Count-Min sketch.[1]

3 OpenSketch Data Plane

In this section, we first describe the design of OpenSketch data plane. We want to be generic and support various measurement tasks but we also want to be efficient and save switch memory (TCAM, SRAM) and use only a few simple hash functions. Next, we discuss how to implement such a data plane with commodity switch hardware at line rate. Finally, we use several example sketches to show how to configure the simple data plane to meet different requirements.

3.1 Generic and efficient data plane

A measurement data plane consists of two functions: picking the packets to measure and storing/exporting the measurement data. OpenSketch allows more customized and efficient ways to pick which packets and which data to measure by leveraging a combination of hashing and wildcard rules. OpenSketch also allows more flexible collection of measurement data by breaking the tight bindings between flows and counters. It reduces the amount of data to store and export using more compact data structures.

Picking the packets to measure: OpenSketch shows that a combination of hashing and classification can support a wide variety of ways of picking which packets to measure. Hashes can be used to provide a compact summary of the set of flows to measure. For example, to count the traffic to a set of servers, we can use hashing to provide a compact representation of the set of servers (e.g., Bloom filter). To count the number of redundant packets with the same content, we can hash on the packet body into a short fingerprint rather than store and compare the entire packet body every time. Hashes also enable a provable accuracy and memory tradeoff.

Classification is also useful for focusing on some specific flows. For example, a cloud operator may need to identify the popular flows from a specific tenant or identify the DDoS attack targets within a list of web servers. Therefore, we need a classification stage to measure different flows with different number of counters or with different levels of accuracy. For example, if there is too much traffic from the IP address 192.168.1.1, we can filter out packets from 192.168.1.1/32 i.e. use one counter to count traffic from the specific IP and another to count remaining traffic of interest from the subnet 192.168.1.0/24. For classifying flows, we can specify wildcard rules that match packets on flow fields and allow some bits in the flow fields to be "don't care". For example, the rule can match packets on source IP prefix 192.168.1.0/24, where the lower 8 bits are "don't care" bits.

Storing and exporting the data: OpenSketch uses a small table with complex indexing. Each entry in the table only contains the counters without the flow fields. These counters can be associated with different entities like a microflow, a wildcard flow, or even a hash value. In this way, OpenSketch allows more flexible data collection with much less memory than traditional flow-based

[1] To insert an element in the reversible sketch, we perform multiple *modular* hash functions on the same key, and increment counters at multiple places. To get the original key, we reverse the modular hash values of the heavy bins, intersect them to get a small set of potential keys. We discuss the technical details of implementing and using reversible sketch in our technical report [44].

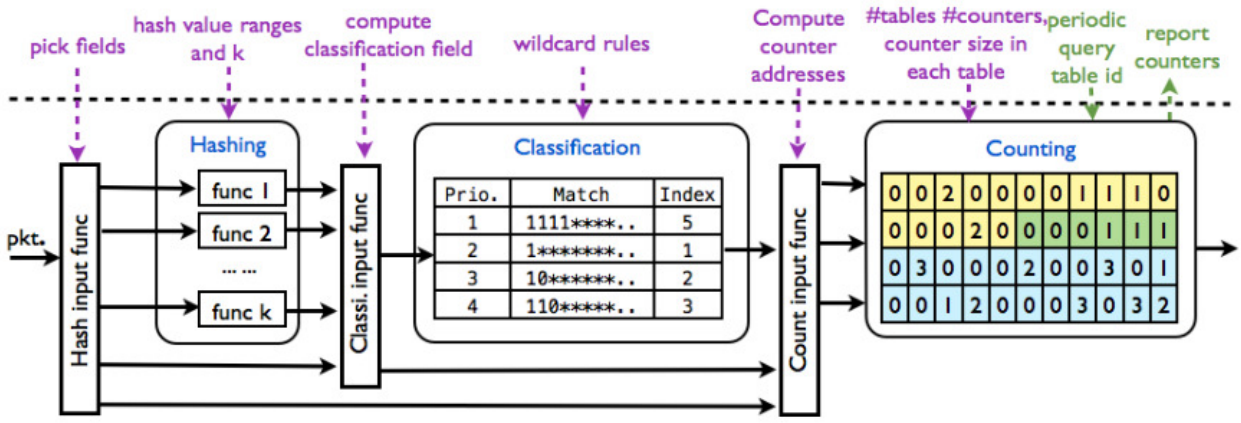


Figure 2: OpenSketch switch data plane

tables. Moreover, OpenSketch can easily export the table to the controller with small bandwidth overhead.

To get such flexibility and memory saving, OpenSketch requires more complex indexing using the hashing and classification modules. Fortunately, these complex indexes are easy to calculate using commodity switch components. For those measurement tasks that need to identify specific flows, the controller can maintain the mappings between counters and flows for classification-based indexing, or reverse engineer the flows from the hash values for hash-based indexing (e.g., using reversible hashing [36]).

OpenSketch data plane: OpenSketch data plane has three stages: a hashing stage to reduce the measurement data, a classification stage to select flows, and a counting stage to accumulate traffic statistics (Figure 2). We use the bitmap as an example to show how the data plane works. Suppose we use the bitmap to count the number of unique source IP addresses to a given destination subnet (say 192.168.1.0/24): First, the hashing stage picks the packet source field and calculates a single hash function. Next, the classification stage picks the packet destination field and filters all the packets matching the rule ($dst : 192.168.1.0/24 \rightarrow 1$). Each rule has an index field, which can be used to calculate the counter location in the counting stage. For example, those packets in the subnet get the index "1", which means counting; the other packets get the default index "-1" and are not counted. Finally, the counting input function calculates the index of the bit to be updated using the hash value of the source field. The corresponding position in the counting table for the bitmap is marked as 1.

3.2 Build on existing switch components

OpenSketch data plane can be easily implemented with commodity switch components:

A few simple hash functions: OpenSketch relies on hashing to pick the right packets to measure with provable memory and accuracy tradeoffs. However, sketches may need a different number of hash functions or different types of hash functions, and may operate on different packet fields. For example, the Count-Min sketch requires k (e.g., 3) pairwise independent hash functions. On the other hand bitmaps [43] and the PCSA sketch [22] require a truly random hash function (i.e. each item is hashed uniformly on its range and completely independently of others).

Fortunately, our experiences of implementing various sketches show that 4-8 three-wise or five-wise independent hash functions are enough for many measurement requirements, and can be implemented efficiently in hardware [34, 39]. Moreover, simple hash functions can make use of the entropy in the traffic to approximate even truly random hash functions well [31].

We can also reduce the number of hash functions by allowing multiple sketches to share the same set of hash functions.

A few TCAM entries for classification: Classification can be easily implemented with high-speed memory TCAMs (Ternary Content-Addressable Memory), which already exist in today's switches to store ACLs (Access Control Lists). TCAMs can match packets with multiple rules *in parallel* and perform actions based on the rule with the highest priority. Each rule contains the matching fields (including 0's, 1's, and"don't care" bits) and actions (such as incrementing a counter and pointing to a SRAM address).

TCAMs are expensive and power hungry, and thus only have a limited number of entries in most switches (e.g., at most thousands of entries [16]). Since OpenSketch leverages a combination of hashing and classification to select packets, it does not maintain individual flow entries in the TCAM and thus significantly reduces the number of TCAM entries to support most measurement tasks. In addition to simple wildcard matching on

packet fields, we allow TCAM entries to match on hash values and leverage the "don't care" bits to perform other operations such as set checking. We will show our detailed design for supporting different sketches in the next subsection.

Flexible counters in SRAM: We store all the counters in the SRAM, because SRAMs are much cheaper, more energy-efficient, and thus larger than TCAMs. Leveraging the provable memory-accuracy tradeoffs in sketches, we can make sure that the sketches always fit in the switch SRAM independent of the traffic pattern.

However, different sketches require different numbers and sizes of counters: the bitmap contains an array of 0's and 1's, while the Count-Min sketch [13] contains several counters for a single packet. We introduce a list of *logical* tables in the SRAM (e.g., in Figure 2 we show three logical tables of different colors). These tables can represent different counters in the same sketch (e.g., k tables for the Count-Min Sketch) or different sketches in the same SRAM. Since all the counters stored in the SRAM can be easily accessed by their addresses, we use a single physical address space to identify the counters in all the logical tables. Based on the table *id* and the relative counter position in the table, we can easily locate the counter position in the SRAM. On those SRAMs that have 6-8 read/write ports [17], these counters can even be updated at the same time.

3.3 Supporting diverse sketches

OpenSketch data plane can support a wide variety of sketches by leveraging a smart combination of hashing, classification, and counting. Here we only discuss a few sketches, for which an implementation in OpenSketch is not obvious. We show more sketch examples such as hash tables, Count-Min sketches, and wildcard filters in [44]. The goal of this subsection is not to show the detailed tricks of implementing a specific sketch, but to show the power of the simple OpenSketch data plane in implementing many complex sketches.

Bit checking operations: Many sketches require more complex bit checking operations than simply comparing the packet fields to a pre-defined rule. For example, the Bloom filter, which is used to filter packets based on a predefined set, requires checking if k positions are 1 or not (based on k hash values calculated from the packets). The DFAs (Deterministic Finite Automaton) [10, 38] and regular expressions [30], which are often used for network intrusion detection, require converting one packet state to another.

To implement such sketches in OpenSketch data plane, we can leverage the hashes and the wildcard bits in TCAMs. For example, to check if a packet's source port belongs to a predefined set of source ports with Bloom filters, we first calculate the Bloom filter array B of 0's and 1's (e.g., 0001101101) of the pre-defined set. Next, we calculate the k hash functions on each incoming packet's source port and generate the packet's array P (e.g., 0001000001). Now we need to check if all the 1's positions in P are also 1's in B. Although such complex bit checking is not supported by TCAM, we can match P with B^*, where B^* replaces all the 1's in B with $*$ (e.g., B^* =000**0**0*) [23]. Then we can match P against B^*. The 0 in B^* correspond to bits that were not set by any packet in B. If P has a 1 where B^* has a 0, then we can conclude that P is not in B. But if P matches the TCAM entry B^*, there is a chance that P is in B, and we say the packet matches the Bloom filter.

Picking packets with a given probability: Many sketches require picking packets with different probabilities. For example, the PCSA sketch (Probabilistic Counting with Stochastic Averaging) [22] provides a way to count the number of unique values of a header field(s) The basic idea is to sample packets into different bins with power-of-two ratios (e.g., 1/2, 1/4, ...$1/2^n$). If there's a packet that falls in the bin i, then that means there are at least 2^i different values. Other streaming algorithms may require sampling packets at a given rate (not necessarily power-of-two) to reduce the measurement overhead.

To support this probabilistic packet selection using only simple uniform hash functions in OpenSketch, we choose to combine these hashes with a few TCAM entries. For example, to implement power of two probabilities, we first calculate the hash value for a packet field using the uniform hash function. Next, we compare the hash value with TCAM rules such as $R_1 : 0***... \rightarrow 1$; $R_2 : 10**... \rightarrow 2$; $R_3 : 110*... \rightarrow 3$; etc. There is a 1/2 chance that a hash value matches rule R_1, 1/4 for R_2, an so on. We can further combine these rules to implement other more complex probabilities.

Picking packets with different granularity: Many streaming algorithms (such as flow size distribution [27], counting distinct destinations with the multi-resolution bitmap [21], and latency tracking with arbitrary loss [26]) often make tradeoffs between flow space coverage and accuracy. If they cover a smaller flow space, they can devote more counters to more accurately measure the covered space, but the measurement result is less representative for the entire flow space. If they cover a larger flow space, the result is more representative but less accurate. Therefore, many algorithms require different levels of coverage in the flow space. For example, the algorithm to measure flow-size distribution [27] hashes on the packets and maps the hash value to different sizes of ranges ([0,1/2],[1/2, 3/4]), ...) and measures

the flow sizes within these ranges. The multi-resolution bitmap [21] uses a similar idea to count the number of unique elements in each range. However, all these algorithms assume customized hardware.

Instead, we design a new multi-resolution classifier that can be easily implemented with TCAM and used in different measurement programs. The multi-resolution classifier uses $\lg(n)$ TCAM entries to represent different ranges in the flow space, where n is the range of the flowspace: $R_1 : 0***... \rightarrow 1$; $R_2 : 10**... \rightarrow 2$; $R_3 : 110*... \rightarrow 3$; etc. Operators simply need to glue the multi-resolution classifier before their other measurement modules to increase the accuracy of their measurements with a good flow space coverage.

4 OpenSketch Controller

With OpenSketch providing a simple and efficient data plane, operators can easily program measurement algorithms and even invent new measurement algorithms by gluing different sketches together. The OpenSketch controller provides a sketch library with two key components: A sketch manager that automatically configures the sketches with the best memory-accuracy tradeoff; and a resource allocator that divides switch memory resources across measurement tasks.

4.1 Programming measurement tasks

Although OpenSketch is sketch-based, it can support a wide variety of measurement tasks because: (1) There are already many sketches for tasks ranging from counting a set of flows, measuring various traffic statistics, to identifying specific flows. (2) Even when some traffic characteristics are not directly supported by sketches, we can still install simpler sketches and implement the complex data analysis part in software in the controller.

As shown in Table 1, we have implemented several measurement programs by simply gluing building blocks together. The algorithms to measure flow size distribution and to count traffic leverage different sketches (e.g., multi-classifier sketches and Bloom filters) to pick the right packets to measure. The heavy hitter detection and superspreader/DDoS detection algorithms leverage Count-Min sketches and k-ary sketches to count specific traffic characteristics. Although there are no sketches that directly measure traffic changes or flow size distribution, we can still rely on sketches to get basic counters (e.g., flow size counters) and leave it to the controller to analyze these counters (e.g., calculate the distribution).

We take the superspreader/DDoS detection as an example to show how to use OpenSketch to implement a measurement task. Superspreaders are those hosts that send packets to more than k unique destinations during a time interval. The goal of superspreader detection is to detect the sources that send traffic to more than k distinct destinations, and ensure that the algorithm does not report the sources with $\leq k/b$ destinations, with high probability $\geq 1-\delta$ [40]. The streaming algorithm for detecting superspreaders [40] samples source-destination pairs and inserts them into a hash table, which requires a customized ASIC to implement in the data plane.

Combining Count-Min sketch and bitmap. Given the building blocks in the OpenSketch measurement library, we implement this superspreader algorithm using a combination of the bitmap, Count-Min sketch, and reversible sketch. Ideally, we would like to use a sketch to count the number of *unique* destinations for each source. However, the Count-Min sketch can only count the number of packets (not unique destinations) for each source, while the bitmap can only provide a single count of the total number of unique destinations but not individual counters for each source. Therefore, we combine the Count-Min sketch and the bitmap to count the number of unique destinations that each source sends: We replace the normal counters in Count-Min sketch with bitmaps to count the unique number of destinations instead of the number of packets. Since the superspreaders algorithm requires us to report a source if the number of retained destinations is more than r, we need a bitmap that can count up to around r destinations. Similarly, we combine reversible sketches with bitmaps to track the superspreader sources.

Sampling source-destination pairs to reduce memory usage: One problem of simply using the sketches is that the trace can still have a large number destinations, leading to large memory requirement to get a reasonable accuracy. To reduce memory usage, we choose to sample and measure only a few source-destination pairs. Similar to the streaming algorithm [40], we sample the packet at rate c/k, and report all source IPs that have more than r destinations, where c and r are defined as a constant, given b and δ as shown in Figure 2 in [40]. For example, when we set $b = 2$ and $\delta = 0.2$, we get $r = 33$ and $c = 44.83$.

Although we use the same sampling solution as the work in [40], there are three key differences: (1) We use Count-Min sketches instead of hash tables, which reduces memory usage with a slightly higher error rate. Count-Min sketches are especially beneficial when there are only a few heavy sources (contacting many destinations) and many light sources (contacting a few destinations), because there are fewer collisions between the heavy and light sources. (2) the work in [40] uses 32 bits to store destination IPs for each pair of source and destination while the bitmap stores only $O(r)$ bits totally. If we set a threshold $r = 33$, we only need a bitmap of

Measurement Tasks	Definitions	Building Blocks
Heavy Hitters [13]	Identify large flows that consume more than a fraction T of the link capacity during a time interval	*Count-Min sketch* to count volume of flows, *reversible sketch* to identify flows in Count-Min with heavy counts
Superspreader/DDoS	A k-superspreader is a host that contacts more than k unique destinations during a time interval. A DDoS victim is a host that is contacted by more than k unique sources.	*Count-Min sketch* to estimate counts for different sources, *bitmap* to count distinct destinations for each *Count-Min sketch* counter, *reversible sketch* to identify sources in *Count-Min sketch* with heavy distinct counts
Traffic changes detection [36]	Identify flows whose absolute contribution to traffic changes are the most over two consecutive time intervals	*k-ary sketch* and *reversible sketches* for consecutive intervals to identify heavy changes
Flow size dist. [27]	Indicate the portion of flows with a particular flow size	*multi-resolution classifier* to index into *hash table* at right granularity; *hash table* to count volume of flows mapped to it
Count traffic	Count the distinct number of source addresses who send traffic to a set of destinations	*Bloom filter* to maintain the set of destinations; *PCSA* to maintain distinct count of sources

Table 1: Implementing measurement tasks using the OpenSketch library

```
for (index in rev_sketch.bins) {
    count = distinct_count.get_count(index, rev_sketch)}
flow_array = rev_sketch.get_heavy_keys(counts, r)
for (flow in flow_array) {
    index_list = count_min.get_index(flow)
    count_list = distinct_count.get_count(index_list, count_min)
    if (all count in count_list > r) output key}
```

Figure 3: The querying function in superspreader detection

149 bits to store all pairs of sources and destinations for each Count-Min counter. (3) It is hard to implement the original streaming algorithm [40] with commodity switch components. This is because the algorithm requires customized actions on the hash table (e.g., looking up the src-dst pair in the hash table, inserting a new entry if it cannot find one and discarding the packet otherwise, removing entries if the hash table is full, etc.).

Querying in the control plane. Figure 3 shows the data analysis part of the superspreader detection program. We first look at the reversible sketch to identify the sources with counts more than r using the distinct counts from the bitmaps and compile a list of heavy bins. The reversible sketch "reverse-hashes" these bins to identify potential superspreader sources. The reversible sketch uses modular hash functions and doesn't provide any accuracy guarantees for the sources. So we next query the Count-Min sketch to verify the counts of the candidate sources. If the count is above the threshold r, then we report the source as a superspreader.

Although we have shown several examples that operators can easily program in OpenSketch, there are still many open questions on how to provide full language support for many other measurement programs. We leave this for future work.

4.2 Automatic config. with sketch manager

Picking the right configurations in the measurement data plane is notoriously difficult, because it depends on the available resources at switches, the accuracy requirements of the measurement tasks, and the traffic distribution. To address these challenges, OpenSketch builds a sketch manager: The sketch manager automatically picks the right sketch to use given the measurement requirements and configures the sketches for the best accuracy given the provable memory-accuracy tradeoffs of individual sketches and the relations across sketches. Furthermore, the sketch manager may automatically install new sketches in the data plane to learn traffic statistics to better configure the sketches.

We take the superspreader problem as an example:

Picking the right sketch for a function. Given the provable tradeoffs between error rate and memory size, the sketch manager automatically picks the right sketch if there are multiple sketches providing the same functions. In superspreader detection, given memory size m bits and the threshold r, there are two common data structures for distinct counters: the PCSA whose error rate is $0.78\sqrt{\lceil \log r \rceil / m}$, and the bitmap whose error rate is $\sqrt{(e^{r/m} - r/m - 1)/(r^2/m)}$. The operator can simply use "distinct counter" as the virtual sketch in his program. The sketch manager can automatically pick the right sketch of distinct counters to materialize the virtual sketch. For example, if there are $m = 149$ bits for distinct counters and $r = 33$, the error rate is 14.5% for PCSA, and 6% for bitmap. Therefore, the manager picks bitmap for the superspreader problem.

Allocating resources across sketches. In OpenSketch, operators can simply configure the sketches required for their programs without detailed configurations. The sketch manager can automatically allocate

resources across sketches within the measurement program, given the memory-error tradeoffs of individual sketches. For the superspreader example, the manager automatically formulates the following optimization problem: configuring the sketch parameters (the number of bits in each bitmap m, the number of hash functions H, and the number of counters for each hash function M in the Count-Min sketch), to minimize the error rate of the superspreader detection, given the total memory size C_{ss}, the threshold r, the sampling ratio c/k, and the estimated number of distinct src-dst pairs N.

$$Min \quad \varepsilon_{ss} = (\varepsilon_{bm} r + \varepsilon_{cm} \times N \times c/k)/r \tag{1}$$

$$s.t. \quad \varepsilon_{bm} = \sqrt{(e^{r/m} - r/m - 1)/(r^2/m)} \tag{2}$$

$$\varepsilon_{cm} = e(1 + \varepsilon_{bm})/M_{cm} \tag{3}$$

$$C_{ss} = H_{cm} M_{cm} m \leq C_{total} \tag{4}$$

$$m_{fillup} \leq m \leq m_{max} \tag{5}$$

Eq.(1) describes the goal of the optimization problem, which is to minimize the error rate of the combination of Count-Min Sketch and bitmap, which is at most $(\varepsilon_{bm} r + \varepsilon_{cm} \times N \times c/k)/r$ for counts $< r$ as proved in [24]. Eq.(2) and Eq.(3) define the error rate of the bitmap and the Count-Min sketch. Eq.(4) describes the memory constraints: The combination of Count-Min sketch and bitmap should take less memory than that available at the switch. Eq.(5) bounds the number of bits m used in bitmap. As suggested in [43], we set m_{fillup} to be the minimum m such that $m \geq 5\sqrt{e^{r/m} - r/m - 1}$, which is the minimum size to guarantee the bitmap doesn't fill up with high probability. We also set $m_{max} = 10r$, which is large enough so that the relative error is less than 1% for distinct counts up to $r = 100$.

Note that we do not consider the reversible sketches in the optimization. This is because the reversible sketch usually has a fixed size of 5 hash tables each with 4096 (distinct) counters as suggested by [36]. In our evaluations with real packet traces we found that for $\delta = 0.2$, 2 reversible sketches of 98-bit bitmaps (a total of 0.5 MB) was often large enough to not miss any sampled superspreaders (thus the false negative rate is caused by sampling, same as the streaming algorithm). In addition, the false positives of reversible sketches are not important because we always check with Count-Min sketch which provides more accurate counts.

We can easily solve the optimization problem to configure sketches automatically. Suppose we want to find $k = 200$ superspreaders with $b = 2$ and a probability of at most $\delta = 0.2$ false positives and negatives. We set the sampling rate at $c/k = 44.83/200$ and report sources with at least $r = 33$ distinct destinations retained. Suppose the the maximum number of distinct source destination pairs on the link is $N = 400,000$, and we have 1900KB available for the combination of Count-Min sketch and bitmap. Then solving the optimization problem, we find that we can minimize the error when the bitmap has $m = 46$ bits, the CM has 3 hash functions and 109,226 counters per hash function.

Installing new sketches to learn traffic statistics. As shown above, some memory-accuracy tradeoffs require understandings of traffic characteristics to have a more accurate configurations of the sketches. For example, in the superspreader example, we need to have an estimation of the maximum number of source-destination pairs N. In this case, OpenSketch requires operators to given a rough estimation based on experiences (e.g., operators can simply give n^2 for a network of n nodes.)

The sketch manager can then automatically install new sketches to help understand such traffic characteristics. For example, to count the unique pairs of source and destination, the manager installs a new distinct counter and periodically checks its value. If the difference between the counting result and the original estimation is above a pre-defined threshold, then the manager re-optimizes the resource allocation problem.

Though we described the sketch manager in the context of finding superspreaders, functions like picking the right sketch and allocating resources across sketches (which don't interact in complex ways) can easily be automated for other measurement tasks too.

4.3 Resource allocation across tasks

When OpenSketch supports multiple measurement tasks simultaneously, it is important to optimize the accuracy of the measurement tasks given the limited resources in the data plane (e.g., the number of hashing modules, the TCAM and SRAM sizes). Therefore, we build a resource allocator in OpenSketch which automatically allocates resources across measurement tasks. Operators simply need to specify the relative importance of accuracy among the tasks. For example, to allocate resources across heavy hitter and superspreader detection, operator can simply specify a weight $\beta \in [0,1]$. $\beta = 1$ means operators care most about heavy hitter detection and $\beta = 0$ means the other way.

The challenge is to allocate resources across tasks without considering the detailed implementation of each task. We propose to modularize the resource allocation problem using optimization decomposition. The basic idea is to introduce a price λ to indicate the relative resource usage and then leave it to each task to optimize their own accuracy based on the resource allocation.

For example, operators may want to detect heavy hitters using a Count-Min sketch and to detect superspreaders using a combination of Count-Min sketch and

bitmaps simultaneously. We first formulate the main allocation problem: Given the total switch SRAM memory C_{total}, the resource allocator allocates the memory across the two measurement tasks (Eq. (7)) to minimize the weighted error (Eq. (6)). For Supersprеaders, the error rate ε_{ss} and space used by the building blocks C_{ss} are as described in 1-4. For heavy hitters the error rate is $\varepsilon_{hh} = e/M_{cm}^{hh}$ and the space used is $C_{hh} = H_{cm}^{hh} \times M_{cm}^{hh}$, where H_{cm}^{hh} is the number of hash functions and M_{cm}^{hh} is the number of counters per hash.

$$Min \quad \varepsilon_{all} = (1-\beta)\varepsilon_{ss} + \beta\varepsilon_{hh} \tag{6}$$

$$s.t. \quad C_{ss} + C_{hh} \leq C_{total} \tag{7}$$

$$given \quad \beta, C_{total}$$

Next, we solve the resource allocation problem in a modularized way by first converting the problem into its dual problem:

$$\begin{aligned} Max \quad & \min((1-\beta)\varepsilon_{ss} + \lambda(C_{ss} - C_{max}/2)) \\ & + \min(\beta\varepsilon_{hh} + \lambda(C_{hh} - C_{max}/2)) \\ s.t. \quad & \lambda \geq 0 \end{aligned} \tag{8}$$

We can then easily decompose the dual problem into two sub-problems corresponding to the two tasks superspreader and heavy hitter detection (The two sub problems are omitted for brevity). Iteratively, the master problem sets a price λ for the space used the two tasks. Each task minimizes its objective at the given price, and the master problem updates its price according to how much extra space was used by the tasks (using the subgradient method). When the price eventually converges to the optimal value, we get the resource allocation across the two tasks and the parameter configurations for each task.[2]

In general, the optimization decomposition technique allows the resource allocator to allocate space efficiently between two tasks without any knowledge of how they're implemented, it only needs to know their relative importance to the operator.

5 Prototype Implementation

Figure 4 shows the key components in our prototype: In the data plane, we implemented the three-stage pipeline (hashing, classification, and counting) on NetFPGA; in the control plane, we implemented the OpenSketch library which includes a list of sketches, the sketch manager, and the resource allocator.

[2]Note that such optimization decomposition only works for convex functions. But the superspreader problem is not convex in terms of number of counters M_{cm}^{ss} *and* the bitmap size m. We work around this by iterate the size of bitmap from m_{fillup} to m_{max}, and thus make the problem convex with a fixed m.

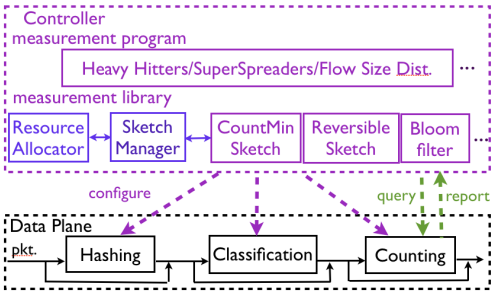
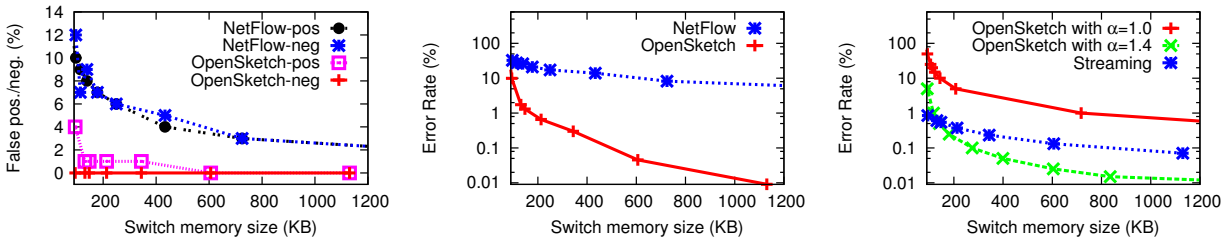


Figure 4: OpenSketch Architecture

Data plane: We implement a OpenSketch switch prototype on NetFPGA to understand its efficiency and complexity in real hardware deployment. We insert our measurement modules into the reference switch pipeline, including Header Parser, Hashing, Wildcard Lookup, and SRAM Counter. Since we simply pull packet headers to collect the statistics without changing the packet bus, there is no effect on packet forwarding latency and throughput. As a packet enters, the Header Parser pulls the related fields from the packet header, which are then hashed by hash functions in parallel. We then pass the packet header and hash values to the Wildcard Lookup, where we implement wildcard rule matching in parallel. For each matched rule, we update the corresponding counters in the SRAM Counter. We use the entire 4.5 MB on-board SRAM to store the counters. To answer the OpenSketch queries from the controller, we implement a userspace software to query these counters via IOCTL calls, which then collects all the counters through PCI interface.

Controller: We implement *seven* sketches in c++ in the controller, including bitmap, PCSA sketch, hash table, multi-resolution classifier, bloom filter, count-min sketch, and reversible sketch. Each sketch has two functions for the measurement programs to use: *configure* to specify the packet fields, the memory constraint, and the number of hash functions to use and *query* to periodically get the statistics. We also implement the sketch manager to configure these sketches and the resource allocator to divide resources across measurement tasks.

The measurement program in the controller can periodically query the sketches about the statistics (e.g., the distinct counts from bitmap, the flows from reversible sketches). According to the received statistics, the measurement programs may install new sketches or change the accuracy requirements accordingly. The sketches automatically queries the data plane to get the counters in order to generate the right statistics to the measurement program. Each time after reporting the counters, the OpenSketch data plane resets the counters as zero in order to monitor the traffic statistics at the next measurement interval. The history statistics are maintained in the controller and written to the disk if necessary.



(a) False positives/negatives of OpenSketch and NetFlow

(b) Error rates of OpenSketch and NetFlow

(c) Theoretical error rates of OpenSketch and a space-saving streaming algorithm

Figure 5: Heavy hitter detection with OpenSketch, NetFlow, and space-saving streaming algorithm

6 Evaluation

To evaluate the balance between generality and efficiency, we use packet-trace driven simulations to compare OpenSketch with NetFlow and other streaming algorithms for several measurement tasks using CAIDA packet traces. To evaluate the feasibility of implementing OpenSketch with commodity switch components, we run stress tests with our OpenSketch prototype. Both our simulator and prototype are available at [5].

6.1 Data plane generality and efficiency

Evaluation setup We build a trace-driven simulator of an OpenSketch implementation of heavy hitters and superspreaders, and compare it to NetFlow with packet sampling, and a superspreader streaming algorithm [40]. We use a one-hour packet trace collected at a backbone link of a Tier-1 ISP in San Jose, CA, at 12pm on Sept. 17, 2009 [41]. We collect the counters at the end of every measurement interval and reset the counters to measure the next interval. We configure the parameters in our evaluation based on recommended values from literature [20, 37]. We set the measurement interval as 5 seconds and run for 120 measurement intervals to ensure the metrics converge.

OpenSketch provides a better memory-accuracy tradeoff than NetFlow. Since both OpenSketch and NetFlow provide general support for diverse measurement tasks, a natural question arises: which one works better? We compare the two using the heavy hitter detection problem because NetFlow, which uses packet sampling, can easily catch large flows and is expected to work well.

We set the threshold T as 0.5% of the link capacity to find a few large senders in the traffic with a heavy-tail distribution. The sketch-based heavy hitter algorithm uses two building blocks: the Count-Min sketch and the reversible sketch, which are described in [44]. These sketches are automatically configured by the sketch manager. We conservatively assume each counter (in both sketches) uses 4 Bytes, though in practice for counting up to $N = 2$ million packets, a counter needs only $lg(N) <$ 3 bytes. For NetFlow with packet sampling, the operator can set the sampling rate according to how much memory is available at the router. We conservatively assume that NetFlow uses 32 Bytes per flow entry (as in [20]) and count the actual number of flow entries used by NetFlow.

Figure 5 (a) shows the false positives and false negatives for identifying heavy hitters. OpenSketch has no false-negatives with 85KB memory, and no false positives when the switch has 600KB memory. In contrast, NetFlow needs 724KB memory to achieve 3% false positives and 3% false negatives. Figure 5 (b) compares the error rate in estimating the volume of detected heavy hitters. Like in [20], we define the error rate as the relative error (the average error of each heavy hitter's volume) divided by the size of the threshold for fair comparison. We can see that with 600KB memory, the error rate for OpenSketch is 0.04%, which is far less than NetFlow.

OpenSketch achieves comparable accuracy to the streaming algorithms, while providing generality. Although OpenSketch supports most sketch-based streaming algorithms, it does not support those streaming algorithms that require complex actions in the data plane. We compare our OpenSketch-based algorithm with these streaming algorithms to understand the tradeoff between accuracy and generality.

Heavy hitter detection: We first compare the two theoretical bounds of OpenSketch-based heavy hitter algorithm and the space-saving streaming algorithm proposed in [8].[3] Figure 5 (c) shows that to achieve the same 0.5% error rate, OpenSketch requires 1354KB memory while the streaming algorithm only takes 160KB. However, OpenSketch can further improve its accuracy when it understands the traffic skew (i.e., the Zipfian parameter α in Sec 2) from either operators or by installing a hash table (similar to the flow size distribution problem in Table 1). For example, if OpenSketch knows that the skew

[3] This is the error with 95% confidence.

parameter is $\alpha = 1.4$, it only needs to configure 142KB memory, which is less than the streaming algorithm.

Superspreader detection: We also compare the accuracy of OpenSketch with the one-level superspreader detection algorithm [40] in Figure 6.[4] We set $k = 200$, $b = 2$, and $\delta = 0.05$. We conservatively assume it takes 8 Bytes to store each src-dst pair in a hash table and 6 Bytes to store each source-counts pair in a second hash table The total space used by the two hash tables for the sampled source destination pairs is the memory size (assuming no hash conflicts).

The streaming algorithm requires an average of 1MB memory to achieve a false positive rate of 0.1% and a false negative rate of 0.3%. For OpenSketch, we use the arrows to show the pairs of false positives (+) and false negatives (X) with different size combination of Count-Min sketches and bitmaps calculated by our sketch manager. With the same 1MB memory, OpenSketch achieves 0.9% false positive rate and 0.3% false negative rate. Note that there are only 29 superspreaders on average during each measurement interval and thus the differences between OpenSketch and the streaming is only 1 superspreader at some intervals. OpenSketch can reach the same false positive and false negative rate as the streaming algorithm when it has 2.4MB memory.

Many measurement tasks can be implemented on top of OpenSketch platform with simple controller code and limited data plane resources: We have implemented *five* measurement tasks on top of OpenSketch (Table 2). The implementation is simple because we only configure existing building blocks and analyze the collected data to report the results. The configuration part takes about 10-25 lines of code and the analysis part takes at most 150 lines of code.

We also show the amount of data plane resources we need for each measurement task according to simulations and theoretical analysis. We only need 4-10 hash functions, less than 30 TCAM entries, and tens of megabytes SRAM to support most measurement tasks. For example, the flow size distribution task needs 3 TCAM entries for its multi-resolution classifier to index into one of three hash tables each of which covers a fraction $(\frac{1}{2}, \frac{1}{4}, \frac{1}{8})$ of the flowspace. We can glue the building blocks together to count traffic in different ways. For example, we can count how many distinct senders contact a set of 200 destination address using a PCSA sketch and a Bloom Filter. We need 10 TCAM entries to store the destinations, each entry corresponds to a Bloom Filter for 20 addresses. For packets that pass the filter, PCSA counts the distinct senders (up to $\sim 2^{17}$) using 17 TCAM entries and 1KB of SRAM.

Resource allocation across measurement tasks: Figure 7 shows how OpenSketch allocates memory resources for two measurement tasks: heavy hitter and superspreader detection with different weight β, given the total memory size 4MB. With a lower β, we devote more resources to superspreaders to ensure its low error rate[5]. When we increase the β to 1 (heavy hitters have higher weights), heavy hitter detection gets more memory resources and thus higher accuracy.

6.2 Prototype evaluation

OpenSketch has no effect on data plane throughput: We deploy our NetFPGA based prototype into a Dell inspiron 530 machine (2 CPU cores and 2 GB DRAM). We connect 4 servers to the 4 Ethernet ports on NetFPGA. We first measure the throughput of OpenSketch switch. We set TCP flows across all four 1GE ports on NetFPGA. The OpenSketch prototype switch can achieve full 1GE throughput among four ports with different packet sizes (64, 512, and 1500 Bytes) without any packet losses. This is because the OpenSketch data plane pipeline does not interrupt the forwarding pipeline in the original switch, and the delay of each measurement pipeline component is smaller than packet incoming rate even for the 64Byte packets. As a result, the packets do not need to stay in the queue for measurement processing.

OpenSketch measurement performance is not affected by multiple hash functions, multiple wildcard rules, but is limited by counter updates. We setup a TCP flow between two servers across NetFPGA, and measure the processing delay of collecting statistics from single packet. We vary the number of hash functions from 1 to 8, and set the number of wildcard rules from 32 to 1024, respectively. The average delay is always 104 ns across all settings. This is because both hash functions and wildcard rules are implemented in parallel in hardware. However, the delay is affected by the number of counters we update for each packet. When we update 5 counters per packet in SRAM (which is the maximum number of updates a sketch may need), the processing delay increases to 200ns. This is because our SRAM can only be read and written sequentially. The performance can be improved with when fabrication of 6 to 8 read ports for an on-chip Random Access Memory is attainable with today's embedded memory technology [17].

[4] The paper [40] also proposes a two-level detection algorithm that only works better than one-level algorithm when there are lots of sources contacting only a few destinations. OpenSketch can also introduce another sampling layer as the two-level algorithm to further improve its accuracy.

[5] Note that here we are considering the error rate for superspreader counters. A 10-20% error rate is enough to ensure low false positive/negative rates.

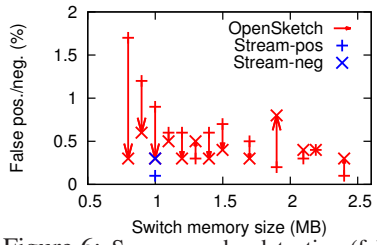


Figure 6: Superspreader detection (false positives (+), false negatives (x))

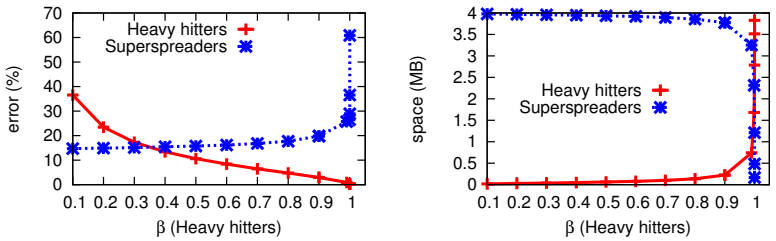


Figure 7: Resource allocation across Heavy hitters and Superspreaders (a) Error rate (b) Memory allocation

Meas. Tasks	Error (%)	Hash func	TCAM entries	SRAM size	Conf LOC	Ana. LOC
Heavy Hitters	0.05 10	3 for CountMin 5 for rev. sketch	0	94KB-600KB (89KB for rev.)	20	25
Superspreaders/ DDoS	0, 0 0.2, 0.2	3 for CountMin,1 for bitmap 5 for rev. sketch	0	0.9MB-1.5MB (0.5MB for rev.)	25	30
Detect traffic changes $>0.5\%$ of total changes	0.1-1	5 for 5-ary sketch, 5 for rev. sketch	0	3.2MB-32MB (82KB for rev.)	20	25
Flow size dist for 100K-2.5M flows	1-2	1	3	300KB-7.5MB	20	150
Count traffic from$\leq$100K src to a set of 200 dst	0.1-1	1 for PCSA, 8 for Bloom Filter	17 for PCSA, 10-16 for B.F.	1KB for PCSA	10	5

Table 2: Implementing measurement tasks in OpenSketch (The numbers for the first two are based on our simulations. The numbers for the later three tasks are based on theoretical analysis in [36, 27, 23, 22]. The error rate is defined as: the relative error for heavy hitters, the false positive and false negative percentages for superspreaders, the relative error for traffic change detection, the weighted mean relative difference in simulation [27] for flow size distribution, and the overall false positive probability of the Bloom Filter for counting traffic (the distinct counter is configured for 10% relative error.))

7 Related Work

Programmable measurement architectures: In addition to NetFlow, there are other works that share our goal of building a configurable or programmable measurement architecture. ProgME [45] allows operators to specify *flowsets* and the switches count the packets in these flowsets. Gigascope [15] is a programmable packet monitor that supports queries on packet streams and automatically splits queries between the data plane and the control plane. In contrast, OpenSketch chooses sketches as the basis of the measurement architecture. Therefore, OpenSketch provides more compact data structures to store statistics with a provable memory-accuracy tradeoff, while supporting a wide range of measurement tasks.

The paper [37] extends NetFlow by using two sampling primitives (flow sampling and sample-and-hold) as the minimalist measurement support in switches, independent of the measurement tasks. Operators can only *passively* process the collected data for their measurement tasks. Other companies [3] build new hardware to provide more line-speed counters at switches. In contrast, OpenSketch allows operators to *proactively* configure different sketches and thus can best use the data plane with guaranteed accuracy for specific measurement tasks. OpenSketch also allows multiple measurement tasks to run at the same time.

Other flexible switch architecture: Software defined networks provide simple APIs at switches and allow the controller to program the switches based on the APIs. PLUG [11] provides flexible lookup modules for deploying routing protocols. OpenSketch shares the same goal of separating the data plane which processes packets, from the control plane that configures how to process the packets. However, existing proposals for software defined networks are not a good fit for measurement tasks. Recent work [16, 32] has recognized the problems of supporting different measurement tasks in OpenFlow [4], such as limited on-chip memory and large communication overhead between the controller and switches. Instead of incremental improvements on OpenFlow, we design a new software defined traffic measurement architecture that provides general and efficient measurement support at switches.

8 Conclusion

Like OpenFlow, which enables a simple, efficient way to control switches by separating the data and control plane, OpenSketch enables a simple and efficient way to collect measurement data. It uses data-plane measurement primitives based on commodity switches, and a flexible control plane so that operators can easily implement variable measurement algorithms. OpenSketch makes sketches more practical by bridging the gap between theoretical research in streaming algorithms and practical measurement requirements and constraints of switches, and makes sketches more flexible in supporting various measurement tasks.

References

[1] http://tools.ietf.org/html/draft-ietf-forces-protocol-22.

[2] http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html.

[3] cpacket. http://www.cpacket.com/download/cPacket_cvu_family_overview_2011a.pdf/.

[4] OpenFlow switch. http://www.openflowswitch.org/.

[5] Opensketch code release. https://github.com/lavanyaj/opensketch.git.

[6] AL-FARES, RADHAKRISHNAN, M., RAGHAVAN, S., HUANG, B., AND VAHDAT, A. Hedera: Dynamic flow scheduling for data center networks. In *NSDI* (2010).

[7] ALON, N., MATIAS, Y., AND SZEGEDY, M. The space complexity of approximating the frequency moments. In *STOC* (1996).

[8] BANDI, N., METWALLY, A., AGRAWAL, D., AND ABBADI, A. E. Fast data stream algorithms using associative memories. In *ACM SIGMOD* (2007).

[9] BAR-YOSSEF, Z., JAYRAM, T. S., KUMAR, R., SIVAKUMAR, D., AND TREVISAN, L. Counting distinct elements in a data stream. In *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques* (London, UK, UK, 2002), RANDOM '02, Springer-Verlag, pp. 1–10.

[10] BREMLER-BARR, A., HAY, D., AND KORAL, Y. CompactDFA: Generic state machine compression for scalable pattern matching. In *INFOCOM* (2010).

[11] CARLI, L. D., PAN, Y., KUMAR, A., ESTAN, C., AND SANKARALINGAM, K. PLUG: Flexible lookup modules for rapid deployment of new protocols in high-speed routers. In *SIGCOMM* (2009).

[12] CORMODE, G., KORN, F., MUTHUKRISHNAN, S., AND SRIVASTAVA, D. Finding hierarchical heavy hitters in streaming data. *ACM Transactions on Knowledge Discovery from Data* (Jan. 2008).

[13] CORMODE, G., AND MUTHUKRISHNAN, S. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms* (2005).

[14] CORMODE, G., AND MUTHUKRISHNAN, S. Summarizing and mining skewed data streams. In *In SIAM Conference on Data Mining* (2005).

[15] CRANOR, C., JOHNSON, T., SPATASCHEK, O., AND SHKAPENYUK, V. Gigascope: A stream database for network applications. In *SIGMOD* (2003).

[16] 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).

[17] DIPERT, B. Special purpose SRAMs smooth the ride. *EDN* (1999).

[18] DUFFIELD, N., LUND, C., AND THORUP, M. Estimating flow distributions from sampled flow statistics. In *ACM SIGCOMM* (2003).

[19] ESTAN, C., KEYS, K., MOORE, D., AND VARGHESE, G. Building a better netflow. *ACM SIGCOMM* (2004).

[20] ESTAN, C., AND VARGHESE, G. New directions in traffic measurement and accounting. *ACM SIGCOMM* (2002).

[21] ESTAN, C., VARGHESE, G., AND FISK, M. Bitmap algorithms for counting active flows on high speed links. In *IMC* (2003).

[22] FLAJOLET, P., AND MARTIN, G. N. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci. 31*, 2 (Sept. 1985), 182–209.

[23] GOEL, A., AND GUPTA, P. Small subset queries and Bloom filters using ternary associative memories, with applications. In *ACM SIGMETRICS* (2010).

[24] HADJIELEFTHERIOU, M., BYERS, J. W., AND KOLLIOS, G. Robust sketching and aggregation of distributed data streams. Tech. rep., Boston University, 2005.

[25] HUANG, G., LALL, A., CHUAH, C.-N., AND XU, J. Uncovering global icebergs in distributed monitors. In *IEEE IWQoS* (2009).

[26] KOMPELLA, R., LEVCHENKO, K., SNOEREN, A., AND VARGHESE, G. Every microsecond counts: Tracking fine-grain latencies with a loss difference aggregator. In *ACM SIGCOMM* (2009).

[27] KUMAR, A., SUNG, M., XU, J., AND WANG, J. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *SIGMETRICS* (2004).

[28] MAI, J., CHUAH, C.-N., SRIDHARAN, A., YE, T., AND ZANG, H. Is sampled data sufficient for anomaly detection? In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement* (New York, NY, USA, 2006), IMC '06, ACM, pp. 165–176.

[29] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling innovation in campus networks. *ACM Computer Communication Review* (Apr. 2008).

[30] MEINERS, C. R., PATEL, J., NORIGE, E., TORNG, E., AND LIU, A. X. Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In *USENIX Security Symposium* (2010).

[31] MITZENMACHER, M., AND VADHAN, S. Why simple hash functions work: Exploiting the entropy in a data stream. In *SODA* (2008).

[32] MOGUL, J. C., AND CONGDON, P. Hey, you darned counters!: get off my asic! In *Proceedings of the first workshop on Hot topics in software defined networks* (New York, NY, USA, 2012), HotSDN '12, ACM, pp. 25–30.

[33] POPA, L., RATNASAMY, S., AND STOICA, I. Building extensible networks with rule-based forwarding. In *OSDI* (2010).

[34] PĂTRAŞCU, M., AND THORUP, M. The power of simple tabulation hashing. *J. ACM 59*, 3 (June 2012), 14:1–14:50.

[35] SANJUÀS-CUXART, J., BARLET-ROS, P., DUFFIELD, N., AND KOMPELLA, R. R. Sketching the delay: tracking temporally uncorrelated flow-level latencies. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference* (New York, NY, USA, 2011), IMC '11, ACM, pp. 483–498.

[36] SCHWELLER, R., GUPTA, A., PARSONS, E., AND CHEN, Y. Reversible sketches for efficient and accurate change detection over network data streams. In *IMC* (2004).

[37] SEKAR, V., REITER, M. K., AND ZHANG, H. Revisiting the case for a minimalist approach for network flow monitoring. In *IMC* (2010).

[38] SHINDE, R., GOEL, A., GUPTA, P., AND DUTTA, D. Similarity search and locality sensitive hashing using TCAMs. In *SIGMOD* (2010).

[39] THORUP, M., AND ZHANG, Y. Tabulation-based 5-independent hashing with applications to linear probing and second moment estimation. *SIAM J. Comput. 41*, 2 (Apr. 2012), 293–331.

[40] VENKATARAMAN, S., SONG, D., GIBBONS, P. B., AND BLUM, A. New streaming algorithms for fast detection of superspreaders. In *Network and Distributed System Security Symposium* (2005).

[41] WALSWORTH, C., ABEN, E., KC CLAFFY, AND ANDERSEN, D. The CAIDA Anonymized 2009 Internet Traces - Sep. 17 2009. `http://www.caida.org/data/passive/passive_2009_dataset.xml`.

[42] WANG, M., LI, B., AND LI, Z. sflow: Towards resource-efficient and agile service federation in service overlay networks. *Distributed Computing Systems, International Conference on 0* (2004), 628–635.

[43] WHANG, K.-Y., VANDER-ZANDEN, B. T., AND TAYLOR, H. M. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst. 15*, 2 (June 1990), 208–229.

[44] YU, M., JOSE, L., AND MIAO, R. Software defined traffic measurement with opensketch. Tech. rep., USC Technical Report, 2013.

[45] YUAN, L., CHUAH, C.-N., AND MOHAPATRA, P. ProgME: Towards programmable network measurement. In *ACM SIGCOMM* (2007).

[46] ZHANG, Y., SINGH, S., SEN, S., DUFFIELD, N., AND LUND, C. Online identification of hierarchical heavy hitters: Algorithms, evaluation, and applications. In *IMC* (2004).

[47] ZHAO, Q., XU, J., AND LIU, Z. Design of a novel statistics counter architecture with optimal space and time efficiency. In *ACM SIGMETRICS* (2006).

[48] ZSEBY, T., ET AL. Sampling and filtering techniques for IP packet selection. `http://www.ietf.org/internet-drafts/draft-ietf-psamp-sample-tech-07.txt`, July 2005. –Work in progress.

V-edge: Fast Self-constructive Power Modeling of Smartphones Based on Battery Voltage Dynamics

Fengyuan Xu* Yunxin Liu† Qun Li* Yongguang Zhang†
*College of William and Mary** *Microsoft Research Asia†*

Abstract

System power models are important for power management and optimization on smartphones. However, existing approaches for power modeling have several limitations. Some require external power meters, which is not convenient for people to use. Other approaches either rely on the battery current sensing capability, which is not available on many smartphones, or take a long time to generate the power model. To overcome these limitations, we propose a new way of generating power models from battery voltage dynamics, called V-edge. V-edge is self-constructive and does not require current-sensing. Most importantly, it is fast in model building. Our implementation supports both component level power models and per-application energy accounting. Evaluation results using various benchmarks and applications show that the V-edge approach achieves high power modeling accuracy, and is two orders of magnitude faster than existing self-modeling approaches requiring no current-sensing.

1 Introduction

Energy consumption is a paramount concern in all battery-powered mobile devices, including smartphones. Power modeling is a key technology and an effective way to understand the power consumption of applications running on mobile devices. Thus, it has attracted much research effort. With a power model, users can identify the power-hungry applications and better manage battery life of their smartphone [1]. Developers are able to profile, and consequently optimize, the energy consumption of their mobile applications [2].

Existing approaches for power modeling have several limitations. First, an accurate power model heavily depends on individual smartphone's hardware/software configuration, battery age, and device usage [3]. Most existing work [4, 5, 6, 7, 8, 9] relies on external power measurement equipment to generate accurate models. This is labor-intensive and requires experts' knowledge. Since the power model of individual smartphone is different and slowly changing [3, 10], it is expensive to apply this approach to build models tailored to every phone. The "self-metering" approach [3, 11, 12] has been proposed to build individualized power models if a smartphone can read the online voltage and current values from its built-in battery interface. While most smartphones have voltage-sensing capabilities, many smartphones today, including popular models like Nexus S and some Samsung Galaxy series, do not have the ability to sense current. Therefore, the previous approach based on current sensing is not applicable to many smartphones. The State-of-Discharge (SOD) approach [13] sidesteps this problem by using the SOD information in battery interface. It does not require current-sensing but has very long model generation time (days) due to the very slow changing nature of SOD. This makes it impossible to have fast power model construction, which is often required to rebuild power models to adapt to various changes in hardware and software, the battery aging, and usage pattern changes (Section 3).

In this paper we propose a new approach for power modeling, called V-edge, to address the limitations of existing approaches. V-edge is self-constructive, does not require current-sensing, and most importantly, is fast in model building. V-edge is based on the following insight to voltage dynamics on battery-powered devices: when the discharge current of a battery is changed, the instant voltage change, caused by the internal resistance, has a reliable linear relationship with the current change. Therefore, from the voltage change, we can determine the change of current and consequently the power information (see more details in Section 4). The V-edge power modeling requires only voltage-sensing and thus works for most smartphones, and is able to generate power models much faster than SOD-based approaches.

We have designed and implemented a power modeling prototype based on V-edge. Our implementation supports both component-level power models and per-application energy accounting. Experimental evaluation results, using various benchmarks and real applications, show that V-edge is able to generate accurate power models, comparable to the power-meter-based approach. The building time is much shorter than SOD-based approaches.

To the best of our knowledge, V-edge is the first work to model smartphone power consumption by leveraging the regularity of instant battery voltage dynamics. Prior to our exploration, these instant dynamics are treated as irregular fluctuations during slow supply voltage dropping (i.e. SOD decreasing) [13]. Our key contributions are as follows.

- We are the first to observe that the current information can be inferred from instantaneous changes in a battery's voltage. We demonstrate that inferring current in such a manner is fast, reliable, and accurate.
- Based on this observation, we propose V-edge to facilitate the self-constructive power modeling on most smartphones. V-edge is much faster than the existing solution, making it efficient to (re)build power models for timely adapting of hardware and software configurations with minimum interruption to users.
- We present the design and implementation of the power modeling system that applies V-edge on popular smartphones, including power models of major hardware components and the per-application energy accounting.
- We evaluate our V-edge-based implementation using a diverse set of benchmarks and applications. The results demonstrate that, given the same model, the error range of the energy estimations of V-edge is within 4%, on average, compared with those of power-meter-based approaches. The model generation is two orders of magnitude faster than SOD-based approaches.

The rest of the paper is organized as follows. In Section 2, we introduce how power modeling works as background. In Section 3, we survey the related work and motivate V-edge. In Section 4, we describe our observation on battery voltage dynamics and demonstrate how to infer current information from voltage readings of battery interface. We present the V-edge energy measurement system in Section 5 and the power models in Section 6. We describe the design and implementation of a system built upon the V-edge power modeling in Section 7 and evaluation results in Section 8. We discuss limitations of V-edge and future work in Section 9 and conclude in Section 10.

2 Background: Power Modeling

A power model estimates the power consumption of a system, such as a smartphone, based on more readily observable system statuses. Typically, the model is generated through a *training phase*. A set of well-designed programs are run to explore various system states in this phase and corresponding power values are measured at the same time. Provided system states and their power measurements as inputs, various modeling techniques like Linear Regression (LR) can derive the relationship between these two sets of information, i.e., a power model.

It is common to simply take resource utilization as the system status, such as screen brightness, CPU usage and so on. As an example of such a *utilization-based* power model, consider a system consisting of only a CPU. To build a power model for this system, one would first design several training programs generating different CPU loads. Then one would run each training program and exploit some measurement tool, like Monsoon Power Monitor [14], to provide corresponding power value P.

Assuming that power consumption of the CPU has a linear relationship with CPU utilization, a power model can be formulated as $P_{cpu} = a * U_{cpu} + b$, where a and b are constant, U_{cpu} is CPU utilization, and P_{cpu} is the estimated power consumption. Here, U_{cpu} is called a *predicator*, as it is used to indicate the power consumption of the CPU. There can be multiple predicators in a power model. For example, if Dynamic Frequency Scaling (DFS) is enabled on the CPU, one may use two predicators, the frequency F_{cpu} and U_{cpu}, to estimate the power consumption. Besides LR, other techniques (e.g., non-linear regression) can also be used to build alternative (often more complicated) power models.

Once a power model is generated, it can be used in a *power estimation phase* to predict the power consumption of the system without requiring additional power measurements. For example, if the CPU utilization of a program is 25% for a duration of T, the the energy consumption of this program is $E_{total} = (a * 25 + b) * T$. More generally, one can monitor and calculate the total energy consumption of a program with dynamic CPU usages as $E_{total} = \sum_i P^i_{cpu} * \Delta T$, where P^i_{cpu} is the i-th measurement of CPU utilization and ΔT is the time interval of the measurement.

Similar to CPU, a power model can be built for other hardware components, such as the screen, Wi-Fi, GPS and so on. After power models of all the components are generated, a power model of the whole system (e.g., a smartphone) can be built on top of the component power models. It is also possible to perform the energy consumption accounting of individual applications or processes, as we will describe in Section 6.

While a power model can give absolute values of energy cost, in practice relative values are often more meaningful to end users. It is usually hard for most users to map absolute energy values (e.g., 10 Joules) to what they concern, such as what percentage of energy has been consumed by screen or an application. As a result, most power monitoring tools on smartphones show power consumption information to users in terms of percentages rather than absolute values [1].

From the above example, we can see that power measurement is the foundation and an essential part of power modeling. As we will see in Section 3, however, the ways in which power measurement is currently done introduces limits to the power modeling's usability and applicability.

3 Related Work and Motivation

System power modeling has been an active research topic and many approaches have been proposed. Based on how power consumption is measured, existing literature can be divided into two categories: *external metering* and *self-metering*. Once power consumption is measured, various *training techniques* to generate models have been studied.

External metering. Most existing work on smartphone power modeling relies on external and expensive power meter to build power models [4, 5, 6, 7, 8]. Those approaches are very accurate because a dedicated power meter can precisely measure power consumption. However, they are labor-intensive and can be done only in a lab. Due to hardware and software diversity of smartphone, each type of smartphones may have a different power model. Any new configuration requires rebuilding the model back in the lab again. Therefore, these in-lab methods are very inflexible and thus not suitable to use in the wild across a large number of users.

Recently, BattOr [9] extended the external meter to mobile settings with a lightweight design. Nevertheless, it is not easy for a layman to operate BattOr because it is not deployed on smartphones. In fact, more and more smartphones use non-replaceable batteries to optimize layout, so attaching any external equipment on them becomes difficult and even dangerous to end-users.

Self-metering. Self-metering approaches [3, 11, 13, 12] collect energy information from smartphones' built-in battery interfaces to generate power models without requiring a power meter. The battery interface consists of battery status registers that the fuel gauge integrated circuit exposes to smartphone operating systems, including voltage, temperature, State-Of-Discharge (SOD), and sometimes current information. The power can be calculated if both voltage and current are provided by the battery interface.

However, many smartphones, including popular ones like the Nexus S and the Samsung Galaxy S2, provide battery interfaces that are only capable of sensing voltages. This means existing self-metering approaches, except [13], are unable to work on a large amount of smartphones (the number is still increasing) already in use.

Zhang et al. [13] proposed building power models based on SOD readings of batteries, which does not require current-sensing. However, the SOD-based approach has a very long model generation time and is inaccurate due to its SOD-based nature. The approach measures the remaining battery capacity (a number from 0% to 100%) to estimate the energy consumption. The granularity of energy measurement is as coarse as 1% of the whole battery capacity. It not only takes tens of minutes to observe a change of battery capacity but also introduces large errors due to the coarse energy granularity.

Motivation of fast power model construction. Fast power model construction is desirable because there are many cases requiring model rebuilding. Besides hardware and software changes, rebuilding is also necessary for changes of software configurations as a simple CPU policy modification may lead to up to 25% differences in power estimation [3]. The battery aging problem [10] also affects power modeling as battery capacity drops significantly with battery age. Thus, a power model needs to be rebuilt after a battery has been used for some time. Furthermore, Dong et al. [3] showed that power models also depend on device usage and demonstrated that a power model should be continuously refined based on usage. In addition, the complexity of modern hardware may require many training cases to generate accurate power models. For example, Mittal et al. [2] used 4096 training cases (for different R, G, B color combinations) to generate a power model for AMOLED display. If it were to take 15 minutes to observe a change of SOD (the minimal time used by Zhang et al. [13]), then it would take the SOD-based approach more than 1,000 hours to generate a single display model, making it almost impossible for end-users to build or rebuild power models on their smartphones.

In addition, the power measuring of a training program need to be performed in a controlled environment. In fast power modeling, short measuring time largely reduces the chance that the user takes the system control back during the running of a training program. Thus, the fast power modeling is more robust because of the tolerance of users' interruptions. Also, the fast one is more flexible because it is able to quickly suspend construction after the completion of a training program and resume later.

Ideally, besides accuracy, a good power model approach should be self-modeling (i.e., it should not depend on external power meters), work for most smartphones (i.e., it should not require current-sensing), and be able to generate models quickly. As shown in Table 1, no existing approach can meet all three requirements. This motivates us to look for a better power modeling approach.

Training techniques for power model construction. Besides LR, other training techniques can also be used for power model construction. For example, Dong et al. [3] used Principal Component Analysis (PCA) to improve the accuracy of a power model by identifying the most effective predicators. Pathak et al. [4] proposed to construct power models using system call tracing. They created Finite State Machines (FSM) for power states of system calls, thus achieving fine-grained power modeling. Our work is complementary to those advanced (and more complicated) model construction techniques. They can be used on top of our battery voltage dynamics based power measurement approach. In this paper, we show that accurate power models can be generated using our new power measurement approach even though we only

Table 1: Comparison of power modeling approaches

	Self-modeling?	Support most phones?	Fast model adaptation?
External metering approaches	✗	✓	✗
Self-metering approaches except SOD	✓	✗	✓
SOD approach	✓	✓	✗
Ideal approach	✓	✓	✓

use basic training techniques for model construction.

4 Sensing Current from Battery Voltage Dynamics

A smartphone is powered by the battery, where supplied voltage is not constant. The voltage dynamics of the battery are exploited here to achieve all desired objectives of the ideal power modeling approach. We show that it is possible to infer discharging current information from instantaneous voltage dynamics of a battery. This inference is reliable enough to be used for power estimation. We also demonstrate that it is practical to detect instantaneous voltage changes by using battery interfaces on smartphones. Based on this, a new energy measurement system is introduced in the next section.

4.1 Battery Voltage Dynamics

The left part of Figure 1 shows the equivalent model of battery electrical circuit [15]. It indicates that at a certain point in time, the voltage reading V of the battery interface can be obtained using

$$V = OCV - V_c - R_b * I$$

where OCV is the open-circuit voltage determined mainly by the remaining capacity of battery, V_c is the voltage drop on the capacitance, R_b is one of the two internal resistors, and I is the discharge current.

When encountering a notable amount of current change, OCV and V_c remain roughly the same value in a short time frame, but the multiplication of R_b and I is sensitive to this current change. As illustrated in the right part of Figure 1, we can observe a sharp edge of voltage readings from the battery interface immediately after the current change. This is known as *internal resistance effect*. After the instantaneous change, the voltage then slowly decreases due to the current discharging on the battery. We define this instantaneous voltage change, $R_b * \Delta I$, as *V-edge*, which is in volts. Clearly, the value of V-edge has a linearly proportional relationship with the change of current. If we measure the V-edge values with the same baseline current I_0 (this can be achieved by starting all the training programs from the same baseline when generating a power model), V-edge has a one-to-one mapping with the current.

$$V_{edge} = R_b * \Delta I = R_b * I - R_b * I_0$$

Or,

$$I = \frac{1}{R_b} * V_{edge} + I_0$$

Via this relationship, we can quickly determine the current value given the V-edge. Next we show that this linear relationship is reliable (Section 4.2) and V-edge can be detected accurately (Section 4.3). Thus, we can use V-edge to further estimate the power consumption and construct power models (Section 5).

4.2 Reliable Relationship between V-edge and Current

The linear relationship between V-edge and current is evident in theory, but because it requires a simplifying assumption about the battery, we seek to understand whether the relationship holds in practice. To this end, we design a set of test trials that run various tasks with different stable workloads on the smartphone. Five batteries for a Google Nexus S phone and three for a Samsung Galaxy Nexus phone were picked for experiments with consideration of different aging stages and manufactures [1]. We ran all tests on these batteries and measured their V-edge values (in μV) respectively. The corresponding current levels (in mA) of these tests were obtained on a Monsoon Power Monitor at a constant voltage level. We then modeled the relationship between V-edge and current using LR for each battery.

Table 2 shows the regression results of eight batteries. The first five batteries are for the Nexus S and the last three are for the Galaxy Nexus. R^2 is the *Coefficient of Determination*, a widely used measure of how well the LR is [16]. We can see that R^2 values of these eight fittings are all above 0.99, indicating very good fitting results. More concretely, Figure 2 shows how well the regression fits the data of battery 2, of which the R^2 value is smallest.

Those real-world experimental results demonstrate that the relationship between V-edge and current is indeed reliable. This provides the foundation of our proposed fast and accurate power modeling approach.

[1] new to one-year-old batteries from four manufactures

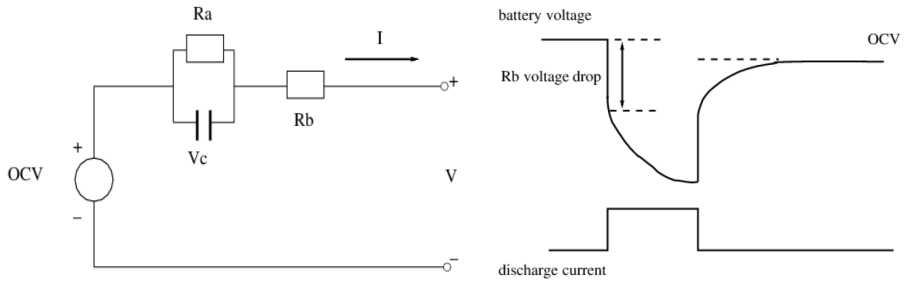


Figure 1: Battery voltage dynamics. Left: Equivalent electrical circuit model for batteries. Right: Battery voltage curve when discharge current is changed. The deep drop of voltage is caused by current increasing on resistor R_b.

Battery	Slope α	Intercept β	R^2
1	0.0048	103.4	0.9987
2	0.0054	100.9	0.9945
3	0.0050	103.3	0.9992
4	0.0054	101.8	0.9991
5	0.0054	102.3	0.9985
6	0.0057	158.9	0.9978
7	0.0056	154.5	0.9979
8	0.0051	157.0	0.9976

Table 2: Linear mapping between V-edge and current on eight batteries of two different smartphones, in the form of current $I = \alpha * V_{edge} + \beta$. R^2 is the metric indicating the goodness of fitting.

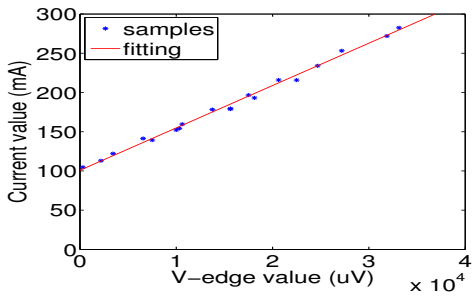


Figure 2: Sampling vs. fitting on battery 2.

4.3 Detecting V-edge

We show in this subsection that V-edge can be easily and accurately captured by battery interfaces on smartphones. Figure 3 illustrates the curve of voltage readings from the battery interface of a Nexus S, when CPU utilization was increased from idle to 95%. We can see a clear voltage drop immediately after CPU utilization (thus the current) was increased. After the instantaneous drop, the voltage decreases very slowly, even with the high discharging current of 95% CPU utilization (the slope will be even gentle if the current draining is smaller). By sampling voltage values from the battery interface before and after the instantaneous voltage change, we can calculate the value of V-edge. Depending on how soon we sample the voltage value after the instantaneous voltage drop, the calculation leads to certain error. Table 3 shows the errors when the sampling happens at different times (i.e., sampling delay) after the instantaneous voltage drop.

We can see that the error is zero if the sample is taken within three seconds of the instantaneous voltage drop. The error increases when the sampling delay becomes larger. If the sampling delay is 10 seconds, the error is 11.76%. Clearly, to reduce error, we should sample voltage value as soon as possible after the instantaneous voltage drop.

The battery interface of smartphones typically updates the voltage value periodically but the updating rate may vary drastically across different phones. For example, the Galaxy S2 updates ten times less frequently than the Nexus S (one update every ten seconds versus one per second). In the case of a low update rate, we should align our voltage sampling with the voltage updating. To achieve this, we employ the following procedure to detect battery interface parameters - the updating interval and time.

We first put the smartphone into idle for a time period longer than its battery interface updating interval (e.g., tens of seconds), then (at time t_0) we increase CPU utilization to a high level and immediately start sampling voltage values at a rate of 1Hz. Once we detect a voltage value change larger than a threshold (i.e., the instantaneous voltage drop caused by increased CPU utilization) at time t_1, we put the CPU into idle again and continue to sample voltage values at 1Hz. When we detect a voltage value change larger than the threshold again (i.e., the instantaneous voltage increase caused by decreased CPU utilization) at time t_2, we stop sampling. Figure 4 de-

Table 3: Sampling error of V-edge in different sampling delays

Sampling delay (s)	1	2	3	4	5	6	7	8	9	10
Sampling error (%)	0%	0%	0%	2.94%	5.88%	5.88%	8.82%	11.76%	11.76%	11.76%

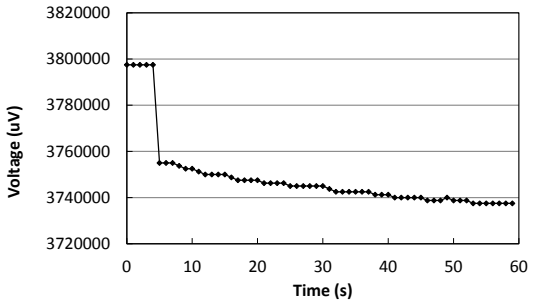


Figure 3: Voltage curve on a Nexus S smartphone when CPU utilization is increased from idle to 95%. Sampling rate is 1Hz.

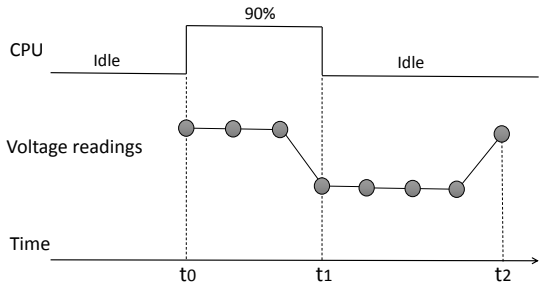


Figure 4: Estimate the voltage updating interval and time of battery interface.

scribes this procedure. Then we treat $\Delta t = t_2 - t_1$ as the updating interval of the battery interface where t_1 and t_2 are the times when voltage updates are triggered. With the sampling rate of 1Hz, the estimation error is within two seconds. Once we know the updating interval and time of the battery interface, we can align V-edge detection with the voltage updating so that the delay of V-edge detection is limited to two seconds. Thus, we can accurately measure the value of a V-edge.

The value of a V-edge is decided by the corresponding current change. If the current change is very small, it is hard to detect the V-edge. To study how likely we can detect a V-edge, we conducted a set of tests with different current changes. Table 4 shows the results. For each current change, we repeated the test 50 times and report the probability that the change was detected. We can see that there is about a 64% chance that the V-edge is detected along with just a 7.5 *mA* increment of current

Current increment (*mA*)	Probability (%)
7.5	64%
15	90%
22.5	98%
30	98%
37.5	100%

Table 4: Probability of capturing current changes

value. With a 30 *mA* change we achieve up to 98% and 100% with 37.5 *mA*. On a Nexus S smartphones, 37.5 *mA* can be caused by a small change of only 4% CPU utilization. To build a power model, we can easily design training programs with a current change much larger than 37.5*mA*. We conclude that V-edge is sensitive enough for component level power modeling.

5 V-edge Energy Measurement System

Once we derive the current information from V-edge, it is feasible to calculate the power information and further generate power models on top of it. Thus, in this section, we show how to build an alternative energy measurement system based on V-edge that is equivalent to the traditional energy measurement systems. In traditional energy measurement systems, the energy cost E of a task is measured by power P and time T, $E = P * T$. Power is decided by current I and voltage V, $P = I * V$. For simplicity, we assume that a task has a constant power consumption during its execution time. The same analysis below can be easily extended to a task with dynamic power consumption by dividing the whole execution time into small time slots with a constant power consumption and using $\sum_i (P^i * T^i)$ to replace $P * T$. That is, the total energy cost $E = \sum_i (P^i * T^i) = \sum_i (I^i * V^i * T^i)$, where i indicates the ith slot.

In our new energy measurement system, we introduce a new term, the V-edge power P_{edge}, to replace the traditional power. The V-edge power is defined as

$$P_{edge} = V_{edge} * V$$

where V_{edge} is the V-edge at the corresponding voltage level V and the unit of P_{edge} is square volts. It does not matter whether the value of V is the voltage value before the instant voltage drop or after the instant voltage drop (see Figure 1) because the difference between the two voltage values is fixed as V_{edge}. That is, the two voltage

values are interchangeable. In our implementation, we choose the voltage value before the instant voltage drop.

Similarly, we define the V-edge energy as

$$E_{edge} = P_{edge} * T = V_{edge} * V * T$$

to replace the traditional energy.

As we show in Section 4.2, V-edge and current have a linear relationship $I = \alpha * V_{edge} + \beta$. Thus, we have

$$\begin{aligned} P &= I * V = (\alpha * V_{edge} + \beta) * V \\ &= \alpha * P_{edge} + P_{edge_0} \end{aligned}$$

where P_{edge_0} is a constant value denoting the baseline V-edge power. That is, we can calculate the real power consumption of a task from the V-edge power of the task. Similarly, we have

$$\begin{aligned} E &= P * T = (\alpha * P_{edge} + P_{edge_0}) * T \\ &= \alpha * E_{edge} + E_{edge_0} \end{aligned}$$

where E_{edge_0} is the baseline V-edge energy.

During power model generation, we can directly measure P_{edge} but not P_{edge_0}. To calculate P_{edge_0}, we employ the following procedure in the power model training phase. We first design two tasks with constant but different stable workloads. We then run the two tasks to consume the same amount of energy in terms of percentage of battery capacity (e.g., 2% of battery capacity which can be achieved by reading SOD information provided by the battery interface). The V-edge power of the two tasks is P^1_{edge} and P^2_{edge}, their traditional power is P^1 and P^2, and their execution time is T^1 and T^2. Without loss of generality, we assume that T^1 is smaller than T^2. As the tasks consume the same amount of energy, we have

$$\begin{aligned} P^1 * T^1 &= P^2 * T^2 \\ (\alpha * P^1_{edge} + P_{edge_0}) * T^1 &= (\alpha * P^2_{edge} + P_{edge_0}) * T^2 \\ P_{edge_0} &= \alpha * \Theta \end{aligned}$$

where $\Theta = \frac{P^1_{edge} * T^1 - P^2_{edge} * T^2}{T^2 - T^1}$ is a known constant value determined by running the two tasks. Note that here we only need SOD readings to derive the value of baseline power, which is done only once. In SOD-based power modeling approaches, every model training program depends on SOD readings, making the model generation time unacceptably long as shown in Section 8.

In fact, even the determination of P_{edge_0} can be skipped if we are only interested in the energy profile excluding baseline, as is usually the case for end users and application developers. Thus, the V-edge energy system is a linear transformation of the corresponding traditional method.

After knowing P_{edge_0}, in the power estimation phase, when a set of tasks run together (e.g., multiple components or processes), we can obtain energy percentage consumed by each task, even without knowing the value of α. For simplicity, let us assume that there are only two tasks, i and j. We can calculate their power percentage from their V-edge power as follows. The energy consumption of task i is

$$\begin{aligned} E^i &= P^i * T^i = (\alpha * P^i_{edge} + P_{edge_0}) * T^i \\ &= \alpha * (P^i_{edge} + \Theta) * T^i \end{aligned}$$

The calculations of task j are similar to task i, so they are omitted due to space limitations.

Thus, the energy percentage of task i is

$$\%E^i = \frac{E^i}{E^i + E^j} = \frac{(P^i_{edge} + \Theta) * T^i}{P^i_{edge} * T^i + P^j_{edge} * T^j + (T^i + T^j) * \Theta}$$

In addition, we can also estimate how long the remaining battery will last. If $X\%$ of battery has been used by tasks i and j, we have

$$\begin{aligned} X\% * C &= E^i + E^j \\ (100 - X)\% * C &= (P^i + P^j) * T^{ij}_L \end{aligned}$$

where C is the battery capacity and T^{ij}_L is the remaining time of the battery if we continue to run both task i and task j at the same time. By solving the above equations, we can get

$$T^{ij}_L = \frac{100 - X}{X} * \frac{P^i_{edge} * T^i + P^j_{edge} * T^j + (T^i + T^j) * \Theta}{P^i_{edge} + P^j_{edge} + 2 * \Theta}$$

If $T^i = T^j = T$, we simply have $T^{ij}_L = \frac{100-X}{X} * T$.

And if we run only task i in the future, the remaining battery time will be

$$T^i_L = \frac{100 - X}{X} * (1 + \frac{P^j_{edge} + \Theta}{P^i_{edge} + \Theta}) * T$$

In summary, the V-edge energy system is able to measure and estimate the power consumption of a system. In the following section, we will develop a system based on V-edge that can address common user concerns such as how much energy a particular application consumes, or how long the battery will last if the user continues running one or more applications.

6 Power Modeling Based on V-edge

We model the power consumption of four major hardware components of smartphones: CPU, screen, Wi-Fi and

GPS. The main purpose is to demonstrate the usability of the V-edge energy measurement system underlying, so we do not introduce new power models. Instead, we use existing or modified ones which are simple and able to capture the main power characteristics of the hardware. In addition, we describe how to do per-application accounting based on generated component-level power models. Power value is provided in the V-edge power $V_{edge} * V$.

CPU Model. DFS is available on most CPUs and often enabled to save power. Thus, for the CPU power model, we consider both CPU frequency and CPU utilization. For each possible CPU frequency, we model the power consumption of the CPU as a linear function of

$$P_{cpu} = a * U_{cpu} + b$$

where U_{cpu} is the CPU utilization.

Screen Model. The power consumption of a screen is decided not only by the brightness of backlight but also by the pixel colors. For example, at the same backlight level of 255, the power consumption of full-screen white is almost three times that of full-screen red.

Dong et al. [17] used RGB values to create a linear model of OLED (Organic Light-Emitting Diode) type displays' power consumption. However, because this model is not suitable for AMOLED (Active-Matrix OLED) types, Mittal et al. [2] proposed another model to also capture the non-linear properties of AMOLED. However, the full model requires 4096 colors, leading to high training overhead. This neutralizes the advantage of self-metering approaches, timely model adaptation. Therefore, we provide a simplified yet effective alternative using the function

$$P_{screen} = f(L) * (c_r * R + c_g * G + c_b * B)$$

where P_{screen} is the screen power consumption, $f(L)$ is a quadratic function of the brightness level L, (R, G, B) is the average RGB value of all pixels, and c_r, c_g and c_b are the coefficient of R, G, and B.

The goal of this screen model is to reduce the number of colors tested. Based on the above function, we derive a preliminary model from only 216 measured RGB colors ($6 \times 6 \times 6$) by first assuming the linear relationship between the power and RGB color. Obviously, this preliminary model does not work well on AMOLED. Additionally, we measure the power of another 125 samples uniformly distributed in the RGB color space. Then we can obtain the power differences of these 125 colors between measured and modeled values. Because the power of AMOLED gradually changes among similar colors, the difference of one in these 125 colors can roughly represent the average offset between measured and modeled power values for all colors nearby. Therefore, the final estimated power value of a RGB color is the calculation of the above function plus the difference of one of the 125 color samples that is closest to this estimated color. In this way, the modeling error decreases to a low level, while a lot of training time is saved.

Note that when we train a screen power model, the power consumption of training programs will include the power consumption of the CPU because the CPU cannot be turned off to run any training program. Thus, we need to remove the power consumption of the CPU from the total power consumption of screen training programs. This is done by generating the CPU power model first and applying it in training screen power model.

Wi-Fi Model. We employ a simple model that considers the data throughput of both directions. A linear power function

$$P_{wifi} = d * D + e$$

is selected where D is the application data, incoming and outgoing, through the Wi-Fi interface. Similar to the screen model, we also remove the power consumption of the CPU in training the power model of Wi-Fi.

GPS Model. We model the power consumption of GPS based on the ON/OFF states, following the work [11, 18]:

$$P_{GPS} = f_{GPS} * S$$

where f_{GPS} is the the power coefficient and S is 1 when GPS is enabled or 0 otherwise.

Per-application Accounting. Users often want to know the power consumption of each individual application so that they can identify where the energy was spent. This per-application power accounting can be done on top of the component-level power models. We can monitor the activities of a process on each component (CPU, screen, Wi-Fi and GPS) and account corresponding power consumption as a function of

$$P_{process} = \sum_i P^i_{cpu} + \sum_j P^j_{screen} + \sum_k P^k_{wifi} + \frac{1}{N} \sum_l P^l_{GPS}$$

where i, j, k, l are the ith, jth, kth and lth time when the process uses CPU, screen, Wi-Fi and GPS, respectively. N is the total number of processes using GPS at the same time. Zhang et al. [13] found that the sum of all component estimates is sufficient to estimate the whole system consumption. Thus, we also adopt this assumption. The power consumption of an application is the sum of the power consumption of all its processes.

7 System Design and Implementation

We have designed a general V-edge-based architecture, illustrated in Figure 5, to run on typical smartphone operating systems. In our design, V-edge runs as a system service in the background, collects data on system resource utilization and activities, generates power models

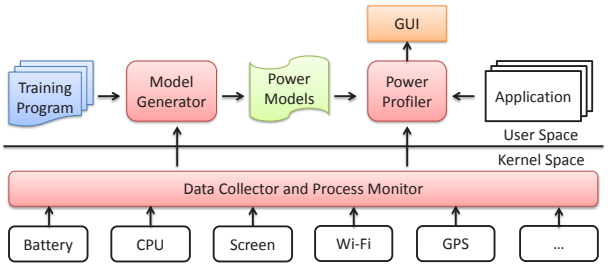


Figure 5: System architecture based on V-edge.

and uses them for power consumption estimation. It also provides a tool with a GUI for users to review the power consumption information of each component and application.

Data Collection. The data collection part is designed to run in the kernel due to two considerations. First, running in the kernel gives us more flexibility and less latency compared with the user space. Second, it avoids the expensive user-kernel mode switching, thus introducing less system overhead. We collect three types of data: voltage readings from the battery interface, utilization information of each hardware component (CPU, screen, Wi-Fi and GPS), and process execution and switching information. For Wi-Fi utilization, we capture the data packets transmitted over Wi-Fi by intercepting the network stack. For process execution and switching information, we hook the kernel scheduler to collect thread scheduling information. We add a new system call to fetch the collected data from the kernel where power model generation and power estimation are done. Voltage information is only used in generating power models, while process information is only used in estimating per-application power consumption. Hardware utilization information is used for both power model generation and power estimation.

Power Model Generation. The system, on top of V-edge, automatically generates component-level power models for CPU, screen, Wi-Fi, and GPS as formulated in Section 6. This is done by running a set of training programs for each component in a controlled way. For example, to build the power model of the CPU, we run CPU training programs with other components in their baseline power states. All training programs of a component run from the same initial state to ensure their measured V-edge values are consistent. For example, each CPU training program starts when the CPU is idle. The model generator also aligns runs of training programs with the voltage updating of the battery interface as we described in Section 4.3, to reduce errors of the voltage sampling. The modeling procedure is done without user awareness when the smartphone is idle and not plugged in. If the user suddenly interrupts the generation by using the phone, the procedure can suspend and resume later with little time penalty, thanks to the short estimation time of V-edge. We also allow power model updates adaptively or through a GUI tool described later in this section.

Power Consumption Estimation. Power estimation is done by tracking hardware resource usage and applying generated models. When users use their smartphones as normal, the data collector keeps running in the background to collect the usage information of each component (frequency and utilization percentage for CPU, brightness level and pixel colors for screen, packet size and number for Wi-Fi and usage of GPS). Thus, the power profiler is able to calculate the power consumption of each component. By tracking process switching, we can know which process is using the resources at a given time. Therefore we can associate resources usage and thus power consumption to the corresponding process, for per-process and per-application accounting.

Power Profiling GUI Tool. On top of the power profiler, we design a GUI tool to show the percentage of the energy consumed by each hardware component and provide a rebuilding option to users.

We have implemented the V-edge-based power modeling and monitoring system on the Android platform. Our implementation in total consists of 2k lines of code for the core components (data collection, model generation and power estimation) and 4k+ lines of code for the training programs.

8 Evaluation

We evaluate our implementation of the V-edge-based system by answering the following questions. 1) How fast can power models be generated? 2) How accurate is the power estimation using the generated models, both at component-level and in per-application accounting? 3) How much system overhead does the implementation introduce in terms of CPU and memory usage?

8.1 Experimental Setup

Devices. We conduct all experiments on a Nexus S smartphone running Android 4.0. We use a Monsoon Power Monitor to measure the actual power consumption of the experiments as the ground truth and for comparison.

Training programs for model generation. We develop a total of 412 training programs to generate power models for CPU, screen, Wi-Fi, and GPS. For each CPU frequency (there are five configurable CPU frequencies on a Nexus S), we use eight training programs with CPU usages randomly picked from eleven possible values (idle to full). Similarly, for the screen we use 347 training programs with different brightness levels and RGB colors

of different pixel blocks. For the Wi-Fi, we use 24 training programs with different packet sizes and transmission rates. Finally, we use one training program for the GPS module.

Benchmarks. We design a set of benchmarks to evaluate the accuracy of our implementation on component-level power estimation. For the CPU we use four benchmarks running for 60 seconds at a CPU frequency of 200 MHz, 400 MHz, 800 MHz and 1000 MHz. For the screen, we use 15 benchmarks. Each of them displays a different picture, as shown in Figure 7, for 10 seconds. For Wi-Fi, we use a benchmark which sends UDP packets with a randomly selected packet size of 50, 100 or 1000 bytes and a random packet inter-arrival time from 1 to 50 milliseconds. The total run time of the Wi-Fi benchmark [2] is 60 seconds. For the GPS, we use a benchmark which uses the location service for 60 seconds.

Applications. We use six real applications to evaluation the accuracy of our implementation on power estimation of real world applications. These applications are *Gallery* where we use the default photo viewer on Android to show 20+ photos (randomly taken by the camera on Nexus S) in slide show mode, *Browser* where we use the default Android browser to read news on Bing News, *Angry Birds* where we play this free version game with commercials, *Video* where we watch a homemade video clip on the default player, *Skype* where we make a VoIP call through Wi-Fi, and *GPS Status* where we run a popular GPS-heavy app from Google Play [19]. Each test performs for one minute.

8.2 Model Generation Time

Model generation time is the time period to run all the training programs and construct power models. It is mainly decided by how quickly power consumption can be measured. In the V-edge approach, as shown in Section 4.3, we can detect the instant voltage changes and consequently measure power consumption in several seconds. However, in a SOD-based approach, power measurement time is much longer because it measures power consumption by observing changes of SOD, at least 15 minutes [13]. Given our 412 training programs, it takes the V-edge-based system for 1.2 hours in total (including the stabilization time between the training cases which can be further optimized) to generate the power models. However, it would take more than 100 hours for the SOD approach to generate the same power models. Our proposed approach is two orders of magnitude faster than the SOD approach.

More importantly, the long model building time of the SOD-based approach demands multiple rounds of the battery recharging, thereby requiring the user intervention. As such, the SOD-based approaches are difficult to automate. With the short modeling time, our approach can be easily done without the user involvement. For the sake of optimization, we can further split the whole procedure into small pieces and manage to complete them one by one. Each piece of modeling tasks just takes minutes of the smartphone idle time and consumes little energy. Thus, the V-edge-based system is transparent to end users.

[2]For the experiment purpose, a stable wireless environment is expected in order to remove the influence of outside factors

8.3 Accuracy

We evaluate the accuracy of the V-edge approach by comparing its energy consumption estimations with both ground-truth measurements and estimations from power-meter-based models. These power-meter-based models are built by measuring power consumption of training programs using an external power meter in the model generation phase. This external-metering approach represents the highest accuracy that one model can achieve because its inputs are precise. Note that the energy comparison is stricter than direct model parameter comparison because model errors can be magnified.

Accuracy of CPU modeling. Figure 6 shows the energy consumption of the CPU benchmarks, including the ground truth and the estimated results of the V-edge approach and power-meter-based approach. Compared to ground truth, the errors of the V-edge approach are 1.45%, 7.89%, 9.71% and 4.18% (5.79% on average). The corresponding numbers of the power-meter-based approach are 1.32%, 5.28%, 5.92%, and 1.54% (3.51% on average). The average difference between our approach and the power-meter-based approach is only 3.65%.

The stable relationship between CPU usage and power consumption introduces small errors to both V-edge-based and power-meter-based approaches.

Accuracy of screen modeling. Figure 7 shows the results of the screen benchmarks. Compared to ground truth, the average error of the V-edge approach is 5.77% (max 15.32%, min 1.22%) and the power-meter-based approach is 5.55% (max 15.81%, min 0.11%). The average difference between our approach and power meter approach is only 3.49%. Note that Figure 7 shows normalized results. The absolute energy consumption of the pictures are very different, as large as 3.3 times.

Our screen model is one of the most sophisticated smartphone screen models considered in self-metering approaches. Nonetheless, our experiments show that it is of limited accuracy (relatively wide error range). The reason is that this model relies on a small number of reference colors to correct initial estimations and provides final answers. Therefore, if a photo has an average pixel color similar to one reference, its estimation error is low. Otherwise, it is a bit high. The model could be optimized,

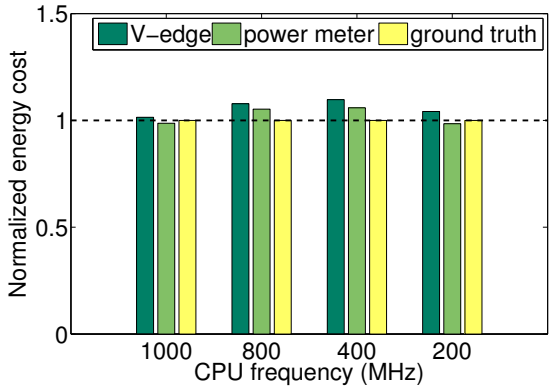


Figure 6: Energy consumption of CPU benchmarks. Results are normalized relative to ground truth values.

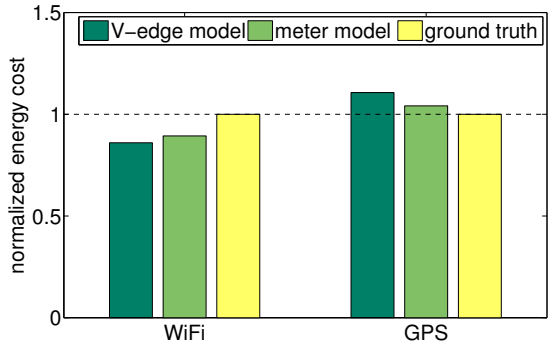


Figure 8: Energy consumption of Wi-Fi and GPS benchmarks. Results are normalized relative to ground-truth values.

but it is out of the scope of this paper.

Accuracy of Wi-Fi modeling. Figure 8 shows the results of the Wi-Fi benchmark. Compared to ground truth, the error of the V-edge approach is 14% and the power-meter-based approach is 10.65%. The difference between two modeling approaches is only 3.75%.

The error of Wi-Fi benchmark is relatively large. This is because our model is simple and served as the comparison platform of two modeling approaches. More predicators like packet numbers per second may improve the accuracy of modeling. Additionally, there are more CPU activities involved in both building and using the Wi-Fi model, compared with other components. Thus, some error is contributed from the CPU model.

Accuracy of GPS modeling. Figure 8 also shows the results of the GPS benchmark. Compared to the ground truth, the error of the V-edge approach is 10.6% and the power-meter-based approach is 4.1%. The difference between our approach and power meter approach is 6.5%.

Accuracy of real applications. Figure 9 shows the results of the six real applications. Compared to the ground truth, the errors of the V-edge approach are 19.5%, 8.6%, 0.2%, 1.6%, 14.7% and 15.5% (10% on average). The corresponding numbers of the power-meter-based approach are 15.6%, 12.5%, 4.2%, 2%, 18.3% and 12.2% (10.8% on average). The average difference between our approach and the power-meter-based approach is only 3.8%.

The accuracy of each component model has an impact on application experiments. For example, the Wi-Fi estimation errors are accumulated quickly in the communication-intensive applications like *Skype*, leading to the relatively large difference between modeled and real results. So is the case of *GPS Status* that has a lot of interactions with the GPS module. As to estimation errors of *Galley*, displayed photos are randomly picked, so it is possible that many of them have average colors not similar to any of 125 references. Another reason is that the average RGB color over all pixels may not be a good predictor. We will investigate this in future. In addition to the individual model accuracy, errors are also introduced by the assumption that the linear combination of all component energy consumption is equal to the whole system consumption. Besides, we do not include power models for other minor energy consumers such as disk I/O.

Summary. All experimental results show that the accuracy of our approach is very close to the power-meter-based approach. The total average difference is only 3.7% for all component-level and application-level power estimations. This demonstrates V-edge's strength in facilitating the self-constructive power modeling.

8.4 System Overhead

Our implementation introduces a very small system overhead in terms of the usage of the CPU and memory. To evaluate, we measured the system CPU and memory usage when V-edge is enabled and disabled for monitoring system energy consumption. With V-edge enabled, the smartphone used only 2 MB more memory to run background V-edge code and store the collect data in memory. Such a small memory footprint is negligible compared with the large memory size of 512MB or 1GB on today's smartphones. We did not observe any noticeable difference on CPU usage because of its event-driven implementation like the work [4]. Thus, our implementation is lightweight and introduces low system overhead.

9 Discussions and Future Work

V-edge provides prior power modeling techniques an opportunity to work on most smartphones on the market. Our power modeling system is one simple example that

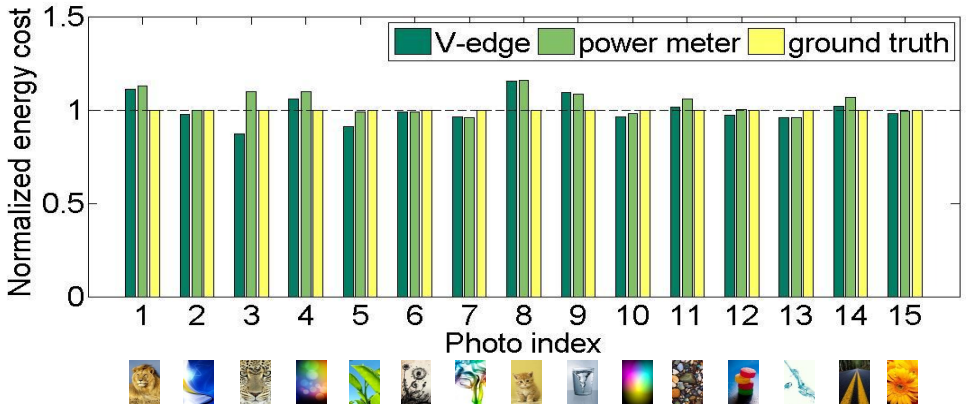


Figure 7: Energy consumption of screen benchmarks. Results are normalized relative to ground-truth values.

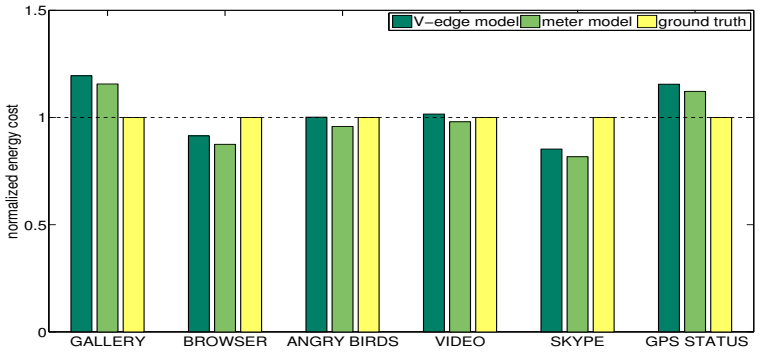


Figure 9: Energy consumption of applications. Results are normalized relative to ground-truth values.

is built upon V-edge. It is intended to demonstrate the implementation feasibility and exhibit benefits that V-edge offers. Therefore, we only cover major energy consumers, such as the CPU and screen. In the future, we plan to complete our models by adding more components, like a 3G module, in order to create a useful system tool.

Another issue worthy of investigation is the model optimization for self-metering approaches. Usually, the more accurate estimations are expected, the more predictors a model need to consider, and thus the more overhead the building procedure has. For example, if we only use the backlight level to model the screen like previous work, 83% building time is saved for our whole system. However, the accuracy is not acceptable. We therefore plan to study how to select more efficient predictors to balance this accuracy and overhead trade-off.

In addition, although our implementation is based on Android platform, the V-edge approach is general enough and not limited to only the Android platform. We plan to implement the V-edge-based system on other mainstream smartphone platforms such as Windows Phone.

10 Conclusions

In this paper, we propose a new approach called V-edge for fast and self-constructive power modeling on smartphones. The V-edge approach is novel because it builds power models by leveraging the regular patterns of the voltage dynamics on battery-powered devices. Different from most existing self-modeling approaches, the V-edge-based approach does not require current-sensing of battery interface so that it works for most smartphones on the market. We have designed and implemented a V-edge-based modeling prototype. It performance demonstrates that V-edge can facilitate fast and accurate power modeling with low overhead.

Acknowledgment. We thank Thomas Moscibroda, Ranveer Chandra, our shepherd Dave Levin, and anonymous reviewers for their valuable comments and insightful feedback. This work was supported in part by US National Science Foundation grants CNS-1117412 and CAREER Award CNS-0747108.

References

[1] Antutu battery saver. https://play.google.com/store/apps/details?id=com.antutu.powersaver&feature=search_result#?t=W251bGwsMSwxLDEsImNvbS5hbnR1dHUucG93ZXJzYXZlciJd.

[2] R. Mittal, A. Kansal, and R. Chandra. Empowering developers to estimate app energy consumption. In *MobiCom*, 2012.

[3] M. Dong and L. Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *MobiSys*, 2011.

[4] A. Pathak, Y. Hu, M. Zhang, P. Bahl, and Y. Wang. Fine-grained power modeling for smpartphones us-ing system call tracing. In *EuroSys*, 2011.

[5] A. Carrol and G. Heiser. An analysis of power consumption in a smartphone. In *USENIX ATC*, 2010.

[6] A. Shye, B. Scholbrock, and G. Memik. Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *MICRO*, 2009.

[7] T. Cignetti, K. Komarov, and C. Ellis. Energy estimation tools for the Palm. In *MSWIM*, 2000.

[8] J. Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *WMCSA*, 1999.

[9] A. Schulman, T. Schmid, P. Dutta, and N. Spring. Demo: Phone power monitoring with BattOr. In *MobiCom*, 2011.

[10] Battery performance characteristics. http://www.mpoweruk.com/performance.htm.

[11] W. Jung, C. Kang, C. Yoon, D. Kim, and H. Cha. Devscope: A nonintrusive and online power analysis tool for smartphone hardware components. In *CODES+ISSS*, 2012.

[12] S. Gurun and C. Krinz. A run-time, feedback-based energy estimation model for embedded devices. In *WMCSA*, 2006.

[13] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. Dick, Z. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *CODES+ISSS*, 2010.

[14] Monsoon power monitor. http://www.msoon.com/LabEquipment/PowerMonitor/.

[15] S. Lee, J. Kim, J. Lee, and B. H. Cho. State-of-charge and capacity estimation of lithium-ion battery using a new open-circuit voltage versus state-of-charge. *Journal of Power Sources*, 2008.

[16] R. Myers. *Classical and modern regression with applications*, volume 2. Duxbury Press Belmont, CA, 1990.

[17] M. Dong and L. Zhong. Chameleon: A color-adaptive web browser for mobile oled displays. In *MobiSys*, 2011.

[18] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *USENIX ATC*, 2012.

[19] GPS status. https://play.google.com/store/apps/details?id=com.eclipsim.gpsstatus2&hl=en.

eDoctor: Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones

Xiao Ma[*†], *Peng Huang*[*], *Xinxin Jin*[*], *Pei Wang*[‡], *Soyeon Park*[*], *Dongcai Shen*[*]
Yuanyuan Zhou[*], *Lawrence K. Saul*[*] *and Geoffrey M. Voelker*[*]
[*]*Univ. of California at San Diego,* [†]*Univ. of Illinois at Urbana-Champaign,* [‡]*Peking Univ., China*

Abstract

The past few years have witnessed an evolutionary change in the smartphone ecosystem. Smartphones have gone from closed platforms containing only pre-installed applications to open platforms hosting a variety of third-party applications. Unfortunately, this change has also led to a rapid increase in *Abnormal Battery Drain (ABD)* problems that can be caused by software defects or misconfiguration. Such issues can drain a fully-charged battery within a couple of hours, and can potentially affect a significant number of users.

This paper presents eDoctor, a practical tool that helps regular users troubleshoot abnormal battery drain issues on smartphones. eDoctor leverages the concept of *execution phases* to capture an app's time-varying behavior, which can then be used to identify an abnormal app. Based on the result of a diagnosis, eDoctor suggests the most appropriate repair solution to users. To evaluate eDoctor's effectiveness, we conducted both in-lab experiments and a controlled user study with 31 participants and 17 real-world ABD issues together with 4 injected issues in 19 apps. The experimental results show that eDoctor can successfully diagnose 47 out of the 50 use cases while imposing no more than 1.5% of power overhead.

1 Introduction

Smartphones have become pervasive. Canalys reported [12] that 487.7 million smartphones were shipped in 2011 — marking the first time that smartphone sales overtook traditional personal computers (including desktops, laptops and tablets).

Configured with more powerful hardware and more complex software, smartphones consume much more energy compared to feature phones (low-end cell phones with limited functionality). Unfortunately, due to limited energy density and battery size, the improvement pace of battery technology is much slower compared to Moore's Law in the silicon industry [40]. Thus, improving battery utilization and extending battery life has become one of the foremost challenges in the smartphone industry.

Fruitful work has been done to reduce energy consumption on smartphones and other general mobile devices, such as energy measurement [8, 13, 39, 46], modeling and profiling [18, 36, 46, 52], energy efficient hardware [21, 30], operating systems [7, 10, 15, 29, 42, 49, 50, 51], location services [14, 20, 26, 31], displays [5, 17] and networking [4, 6, 32, 41, 43]. Previous work has achieved notable improvements in smartphone battery life, yet the focus has primarily been on normal usage, i.e., where the energy used is needed for normal operation.

In this work, we address an under-explored, yet emerging type of battery problem on smartphones – *Abnormal Battery Drain (ABD)*.

1.1 Abnormal Battery Drain Issues

ABD refers to abnormally fast draining of a smartphone's battery that is not caused by normal resource usage. From a user's point of view, the device previously had reasonable battery life under typical usage, but at some point the battery unexpectedly started to drain faster than usual. As a result, whereas users might comfortably and reliably use their phones for an entire day, with an ABD problem their batteries might unexpectedly exhaust within hours.

ABD has become a real, emerging problem. When we randomly sampled 213 real world battery issues from popular Android forums, we found that 92.4% of them were revealed to be ABD, while only 7.6% were due to normal, heavier usage (Section 2). Further, rather than being isolated cases, many ABD incidents affected a significant number of users. For instance, the "Facebook for Android" application (Table 1-a) had a bug that prevented the phone from entering the sleep mode, thus draining the battery in as rapidly as 2.5 hours. The estimated number of its users was more than *12 million* at that time [24], among whom a large portion were likely to have been affected by this "battery bug".

The emerging pervasiveness of ABD issues is a collateral consequence of an evolutionary change in the smartphone industry. In the last few years, a new ecosystem has emerged among device manufacturers, system software architects, application developers, and wireless service carriers. This paradigm shift includes three aspects:

(1) The number of third-party smartphone applications (or "apps" for short) has grown tremendously (Google Play: 600,000 apps and 20 billion downloads [47]; App Store (iOS): 650,000 apps and 30 billion downloads [2]), however, most app developers are not battery-cautious.

ID	Category	App/System	Root Cause	Resolution
(a)	App Bugs	Facebook	The 1.3.0 release (Aug. 3rd, 2010) of this app contained a bug that kept the phone awake.	Downgrade to the previous version.
(b)	App Bugs	Gallery	The user opened a corrupted picture file in "Gallery", which caused the "mediaserver" process to run into an abnormal state and hog the processor.	Automatically terminate the "mediaserver" after the user uses the "Gallery" app.
(c)	App Config	WeatherBug	A configuration change made "WeatherBug" check locations and update weather information more frequently. Heavier usage of GPS causes the battery to drain quickly.	Roll back the configuration changes to less frequent updates.
(d)	App Config	Android Browser	The GPS was continually turned on because the browser was trying to find the location of the user, as requested by "google.com".	Go to "google.com" and disable "Allow use of device location".
(e)	System Bugs	Android System	A bug in the Wi-Fi device driver on Nexus One caused the phone to repeatedly enter its suspend state and immediately wake up, resulting in severe battery drain.	The driver developer has to modify their code to fix the problem.
(f)	System Config	Android System	The user configured the CPU to run at an unnecessarily high frequency.	Roll back the configuration change.
(g)	Environment	Android System	The user's office building contains several radiology devices, which interfere with cell signals and thus make the phone spend more power searching signals.	Turn on Airplane mode when the user is in the office.

Table 1: **Representative ABD examples collected from Android forums.**

Smartphone apps used to be primarily made by device manufacturers, with appropriate training and development resources. In contrast, smartphone apps are now mostly developed by third-party or individual developers. They tend to focus limited resources on app features, on which purchase decisions are often made, but put less effort on energy conservation.

(2) The hardware/software configurations and external environments of smartphones have become diverse, making it difficult and expensive to test battery usage under all circumstances. As a result, many battery-related software bugs escape testing, even by professional software teams, e.g., a bug in Android that affected certain Nexus One phones, (Table 1-e), and a bug in iOS that caused a coninuous loop when sychronizing recurring calendar events [11].

(3) In addition to software defects (e.g., Table 1–a, b, d and e), ABD issues can also be caused by configuration changes (e.g., Table 1–c, f) or environmental conditions (e.g., Table 1–g). In many of such cases, their root causes are not obvious to ordinary users. Therefore, it would be beneficial if the smartphone system itself could automatically diagnose ABD issues for users.

1.2 Are Existing Tools Sufficient?

Existing energy profilers, such as Android's "Battery Usage" utility, PowerTutor [52], and Eprof [36, 35], monitor energy consumption on smartphones. While they provide some level of assistance to developers or tech-savvy users in troubleshooting ABD issues, they are insufficient for broadly addressing ABD issues due to three main reasons:

(1) These tools *cannot differentiate normal and abnormal energy consumption*. A high energy consuming app does not necessarily cause ABD. To determine an app is "normal" or "abnormal", a user needs to know how much battery the app is supposed to consume, which is difficult for typical users, especially since an app's battery usage can fluctuate even with normal usage.

(2) The information provided by these tools requires technical background to understand and take actions on. Even for tech-savvy users, information form these tools are not sufficient for identifying the *ABD causing event* (e.g., an app upgrade). Knowing causing events is critical for pinpointing the right root cause and determining the best resolution.

(3) As mentioned in Section 1.1, sometimes an ABD issue may be caused by the underlying OS, thereby affecting all apps. In this case, these profiling tools may not be able to shed much light on the root cause, much less be helpful to identify a resolution to an ongoing ABD issue.

Apps like JuiceDefender [27] automatically make configuration changes to extend battery life. They help preserve energy during normal usage, but they cannot prevent or troubleshoot ABD issues.

From a user's point of view, a highly desirable solution is to have the smartphone itself troubleshoot ABD issues and suggest solutions with minimum user intervention. Besides helping end users, such systems can also collect helpful clues for developers to easily debug their software and fix ABD-related defects in their code.

1.3 Our Contribution

This paper presents *eDoctor*, a practical tool to help troubleshoot ABD issues on smartphones. eDoctor records re-

source usage and relevant events, and then uses this information to diagnose ABD issues and suggest resolutions. To be practical, eDoctor meets several objectives, including (1) low monitoring overhead (including both performance and battery usage), (2) high diagnosis accuracy and (3) little human involvement.

To identify abnormal app behavior, eDoctor borrows a concept called "phases" from previous work in the architecture community for reducing hardware simulation time [44, 45]. eDoctor uses phases to capture apps' time-varying behaviors. It then identifies suspicious apps that have significant phase behavior changes. eDoctor also records events such as app installation and upgrades, configuration changes, etc. It uses this information in combination with anomaly detection to pinpoint the culprit app and the causing event, as well as to suggest the best repair solution.

To evaluate eDoctor, we conducted a controlled user study and in-lab experiments: **(1) User study**: we solicited 31 Android device users with various configurations and usage patterns. We installed eDoctor and popular Android apps with *real-world* ABD issues on their own smartphones. eDoctor could successfully diagnose 47 out of 50 cases (94%). **(2) In-lab experiments**: we also measured the overhead of eDoctor in terms of its energy consumption, storage consumption and memory footprint. The results show that eDoctor adds little memory overhead, and only 1.24 mW of additional power drain (representing 1.5% of the baseline power draw of an idle phone).

2 Real-world Battery Drain Issues

To understand battery drain issues on smartphones, we randomly sampled 213 real-world battery drain issues from three major Android forums: AndroidCentral.com, AndroidForums.com, and DroidForums.net. To effectively sample the issues from the thousands of battery-related discussion threads in each forum, we searched a set of keywords including "battery", "energy", "drain", and their synonyms, and then randomly picked 213 issues that were confirmed to be resolved. With the collected issues, we studied their root cause categories, triggering events and repair solutions found by users (e.g., removing an app or adjusting configuration) to get guidelines for eDoctor's design.

2.1 Root Cause Categories

We studied the root cause categories and distribution of the problematic components (Figure 1). We made the following observations.

(1) *The majority (92.4%) of the sampled battery life complaints by users are related to abnormal battery drain, and only 7.6% are about heavy yet normal battery usage of some mobile apps*. This breakdown indicates that (i) ABD is an emergent and pervasive problem for smartphones, and (ii) before trouble-shooting a battery issue, one may first need to know whether it is indeed caused by some abnormal problems or if it is simply due to heavy usage of the device or a particular app. An energy profiler can give the battery usage of each app, but cannot usually tell whether the usage is normal or abnormal.

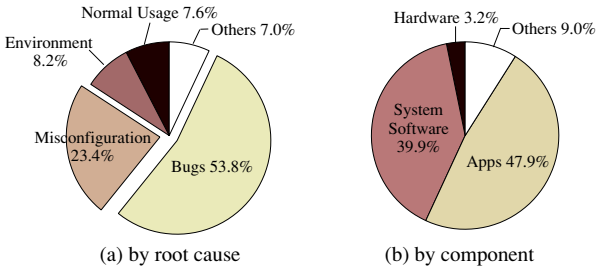


Figure 1: **Distribution of 213 real-world battery drain issues that we randomly sampled**. The meaning of *Others* in each graph: (a) problems with uncommon root causes such as battery indicator error; (b) other sources causing battery drains such as environmental conditions.

(2) *Application issues cause 47.9% of all examined cases*. This observation supports our assertion that app developers are not energy cautious. About three-quarters of the app issues have been identified as app bugs and the remaining are related to configuration. Besides app issues, other factors such as bugs in the system (22.2%), configuration changes (11.8%) and environmental conditions (8.2%) can all lead to ABD issues. It would save a user (even a tech-savvy user) time and effort if a tool can automatically pinpoint the reason for ABD issues and suggest a repair solution accordingly.

(3) *Overusing or misusing certain types of resources can cause ABD issues*. Software bugs and misconfiguration can result in misusing or overusing *certain types of resources*, such as GPS, sensors, etc., leading to an ABD problem. These situations imply that it is beneficial to monitor and analyze usage on those resources. By doing so, eDoctor can separate abnormal from normal battery drains and also suggest detailed repair solutions directly related those resources.

For many ABD issues, especially those caused by misconfiguration and system bugs, it is difficult for an energy profiler to diagnose. For example, enabling background data transmission may result in high energy consumption of certain apps that transfer data when running in the background. Profilers may list these apps as top energy consumer, which mislead users to think they became abnormal and thus remove them.

Events	Cases	Appropriate Solution
App Installation	27	Remove app
App Upgrade	11	Revert to the previous version
System Upgrade	28	Wait for new update
Configuration change	15	Adjust configuration
Environmental change	12	Adjust configuration
Others	16	Others
Not remembered	104	-

Table 2: **ABD-triggering events and the most appropriate resolving solutions.** "Not remembered" refers to the cases where users do not remember what they have done that could possibly cause ABD issues.

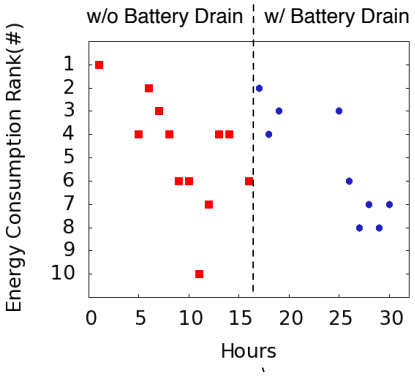


Figure 2: **Battery consumption rank of the Android Gallery app running on a real user's phone**. We recorded the battery consumption rank of this app reported by the Android "Battery Usage" utility, once every hour. The first 15 hours is the time period when the app does not have the battery bug, whereas the second 15 hours is the period when the bug manifested.

2.2 Triggering Events and Resolutions

In general, ABD issues happen only after certain events, e.g., installing a buggy app, upgrading an existing app to a buggy version, changing configurations to be more energy-consuming, entering a weak signal area, etc. Therefore, knowing such triggering events is critical for suggesting appropriate repair solutions to users, as shown in Table 2.

Interestingly, however, in more than 48% of the 213 ABD issues, users did not remember what they had done previously or what could be the possible ABD-triggering event. In such cases, manually diagnosing and resolving the issue becomes difficult. Simply removing a suspicious app, probably the one reported by energy profiling tools as a high energy consumer, is not always the most appropriate solution; it can be either overkill or even incorrect.

3 Execution Phases in Smartphone Apps

To identify the problematic app or system for an ABD issue, it is critical to differentiate abnormal from normal battery usage. It is natural to immediately focus on the app that is the top battery consumer as reported by an energy profiler. Unfortunately, as shown in Figure 2 from a real case, such approach does not always work because an app's rank in the battery consumption report can fluctuate over time. The challenge is that there is no clear difference between normal and abnormal periods. Thus, energy profiles and rank are not reliable indicators for troubleshooting ABD issues. Additionally, Figure 2 shows that *changes* in battery consumption or rank of an app are also not accurate indicators for abnormal behaviors for similar reasons.

To identify abnormal app behaviors, eDoctor borrows a concept called "phases" from previous work for reducing hardware simulation time [16, 19, 23, 28, 38, 44, 45]. The previous work has shown that programs execute as a series of phases, where each phase is very different from the others while still having a fairly homogeneous behavior between different execution intervals within the same phase. Hardware researchers simulate those representative phases to evaluate their design instead of the entire execution [45].

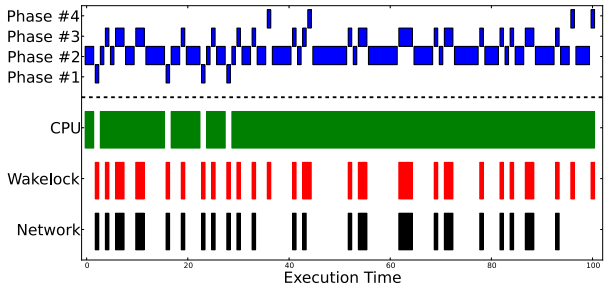
Phase Identification. Inspired by the previous work, eDoctor uses phases to capture an app's behavior in terms of resource usage. The execution of an app is divided into execution intervals, which are then grouped into phases. Intervals in the same phase share similar resource usage patterns. When an app starts to consume energy in an abnormal way, its behavior usually manifests as new major phases that do not appear during normal execution. Combining such phase information together with relevant events, such as a configuration change, eDoctor can identify both the culprit app and triggering event with high accuracy.

Prior hardware simulation work studied architecture related behaviors (e.g., cache miss ratio), so they captured phases based on instruction-level information, such as basic block vector (BBV). However, such fine-grained information is not suitable for identifying resource usage phases because it does not directly correlate to resource usage. Smartphone apps are different from most desktop or server applications — they are usually relatively simple and not computationally intensive, but rather I/O intensive, interacting with multiple resources (devices) such as the display, GPS, various sensors, Wi-Fi, etc. These resources are energy consuming, so mis-using or over-using these resources leads to ABD issues. Therefore, we can identify phases by observing how these resources are used by an app during different execution intervals.

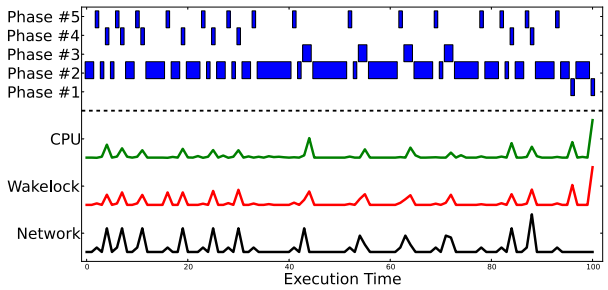
Our first approach starts from a fairly coarse-grain level by recording only resource types used during each execution interval. We refer to this method as *Resource Type Vector (RTV)*. It is based on a simple rationale that different execution phases use different resources. For example,

an email client app uses the network when it receives or sends emails. But when the user is composing an email, it uses the processor and display. The RTV scheme uses a bit vector to capture what resources are used in an execution interval. Each bit indicates whether a certain resource type is used in this interval. If two intervals have the same RTV, they belong to the same phase.

As shown in Figure 3(a) with data collected from the Facebook app used in a real user's smartphone, RTV clearly shows some patterns and phase behaviors: during different phases, different types of resources are used, and phases appear multiple times during different intervals. As the figure shows, the most frequent phase is that only the CPU is running. In this phase, most of the time the app is idle. The second most frequent phase has both CPU and network active, which indicates the app transfers and processes data.



(a) Phase pattern based on RTV



(b) Phase pattern based on RUV

Figure 3: **The phase behavior of the Facebook App in a real user's smartphone**. In the top part of both figures, the shaded bars indicate which phase the app is in. In the bottom part of figure (a), shaded bars indicate the resource is in use. In the bottom part of figure (b), the curves indicate the amount of resource usage.

Although the RTV scheme is simple, it turns out to be too coarse-grained. An app may use the same types of resources in two different phases, but their resource usage rates differ. For example, for an email app, while both the email updating phase and email reading phase use the display, CPU and network, the resource usage rates are different. The former typically has more network traffic. Therefore, we explored a second scheme — *Resource Usage Vector (RUV)*. Each element in a RUV is the amount of usage of the corresponding resource.

We calculate the usage of a resource by the amount of the resource normalized by the CPU time. The execution interval cannot be too small in order to control the measurement overhead, so an app may run for only a fraction of one execution interval. In that case, absolute usage numbers cannot precisely represent the usage behavior. CPU time is a good approximation of the amount of time an app actually runs. Normalizing to CPU time allows us to correlate two intervals that belong to the same phase, even if the app runs for different amounts of time in each interval.

If two execution intervals have similar RUVs, they belong to the same phase. Similar to previous work [45], we use the k-means algorithm to cluster intervals into phases. To find the most suitable k (i.e., the number of clusters to generate), eDoctor tries different k from 1 to 10 at runtime. For each k, we evaluate the quality of the clusters by calculating the average inter-cluster distance divided by the average intra-cluster distance as a score. The higher the score is, the better the clusters fit the data. Since the best k is likely to be the largest k it tries, we pick the smallest k whose score is as high as the 90% of the best score.

Figure 3(b) shows the RUV phase behavior using the same data. As it shows, RUV captures one more phase compared to the phases divided by RTV, enabling eDoctor to further differentiate between low and high network usage. More specifically, phase #3 and phase #4 both have usage of CPU, wakelock and network, but phase #4 has higher network usage. It provides more fine-grained information regarding an application's phase behavior.

4 eDoctor: Design and Implementation

The objective of eDoctor is to help users diagnose and resolve battery drain issues. Even though the information offered by eDoctor can also be used for app developers, our goal is to help users troubleshoot and/or bypass ABD issues before developers fix their code which as shown may take months. Therefore, instead of locating root causes in source code, eDoctor's diagnosis focuses on identifying (1) which app causes an ABD issue and (2) which event is responsible, e.g., the user updated an app to a buggy version or made an improper configuration change. Based on such diagnosis result, eDoctor then suggests appropriate repair solutions.

There are two major challenges involved in achieving these objectives. First, it is non-trivial to accurately pinpoint which app and event accounts for the ABD issue. The causing event may not be the most recent one; instead, it can be followed by many other irrelevant events,

e.g., the case where the user installed a buggy app and then made multiple configuration changes. Second, eDoctor itself should not incur high battery overhead. It needs to balance the energy overhead and the amount of information needed for accurate diagnosis.

This section presents our design of eDoctor. As an overview, eDoctor consists of four major components: Information Collector, Data Analyzer, Diagnosis Engine, and Repair Advisor. The Information Collector runs as a light weight service to collect resource usage and event logs. The Data Analyzer performs phase analysis (Section 3) on the raw data and stores intermediate results to facilitate future diagnosis. Off-line analysis is done only when the phone is idle and connected to external power, in order to avoid affecting normal usage. When users notice ABD, they initiate the Diagnosis Engine to find the culprit app and the causing event. Based on the diagnosis result, the Repair Advisor provides the most relevant repair suggestions.

eDoctor can be installed as a standalone app. It runs on most Android phones and it is compatible with all Android versions since 2.1. A modified Android ROM is optional to track app-specific configuration changes.

4.1 Information Collector

The Information Collector records three main types of data in the background: (1) each app's resource usage, (2) each app's energy consumption, and (3) relevant events such as app installation, configuration, and updates.

Resource usage. eDoctor monitors the following resources for each app: CPU, GPS, sensors (e.g., accelerometer and compass), wakelock (a resource that apps hold to keep the device on), audio, Wi-Fi, and network. To facilitate diagnosis, eDoctor records resource usage in relatively small time periods (called *recording interval*). The default recording interval is five minutes in our implementation.

What resource usage information to store depends on the phase identification method (Section 3). RTV uses a bit vector to record whether the resources have been used in each recording interval. RUV, on the other hand, records the usage amount of each individual resource, e.g., time in microseconds, amount of network data in bytes.

In our implementation, eDoctor takes advantage of the resource usage tracking mechanism in the Android framework. This mechanism keeps a set of data structures in memory to track resource usage of each app. The resource usage data are maintained for each individual app, even if multiple apps run during the same recording interval. The values recorded are accumulated amounts since the last time the phone was unplugged from its charger. At the end of each recording interval, eDoctor reads these values and calculates the resource usage amounts in the past recording interval. Figure 4 shows a simplified example of a resource usage table for an app.

Some resources can be simultaneously accessed by multiple apps without consuming extra energy. For example, once a GPS unit is turned on, it gathers location examples, and it does not consume extra energy if more than one app requests those examples. eDoctor performs coarse-grained accounting of such resources; so if N apps access such a resource for overlapped T time units, each app is charged for T time units of resource utilization. Fine-grained energy profilers like Eprof [35] use a proportional accounting scheme, such that each app would only be charged for T/N units of resource utilization. eDoctor's uses the coarse-grained schema because its goal is to track app-specific energy patterns, not overall energy fluctuations of the whole system.

Energy consumption. In addition to resource usage, eDoctor also records battery consumption of each app in each recording interval. Energy consumption is used for two main purposes: (1) to prune apps with small energy footprints, which are unlikely a cause for ABD, and (2) to rank suspicious apps according to the consumed energy of each app. As we use the battery consumption information only for such comparative purposes, it is less critical to have high fidelity measurement. Further, simple models provide superior performance benefits that are essential to reduce overhead of eDoctor, because it doesn't have to track fine-grained information such as energy state switches. Therefore, we employ an efficient profile-based energy model instead of expensive state-based energy models [46, 52].

Each Android device comes with data about power consumption of various hardware components measured by the manufacture, e.g., the average power consumption of the processor running at different frequencies and the average power consumption of the Wi-Fi device being idle or sending data. eDoctor combines this average power consumption data by the usage data it collects to estimate the total energy consumption of an app during each recording interval. This energy model has been used in both industry (e.g., Android's "Battery Usage" utility [1]) and academic research (e.g., ECOSystem [51]).

Events. Events are critical for both diagnosis and repair advisory. eDoctor records two types of events: (1) configuration changes, and (2) maintenance events (installation, updates). Such events may be initiated not only by the users, but also by the underlying system automatically. App and system configuration entries and their new values are recorded as key-value pairs. Since most apps use Android's facility components (e.g., `SharedPreferences`) to manage configurations, we track app configurations by modifying these common components. `SharedPreferences` is a general framework that allows developers to save and retrieve persistent key-value pairs of primitive data types, which is suit-

able for managing user preferences. We modified the implementation of the `SharedPreferences.Editor` interface to let it send a broadcast message to eDoctor whenever a preference entry is changed. Each message contains the name of the app, the preference file name, the preference key name and its new value. These messages are identified with a special key and only eDoctor can receive them, so they are effectively unicast messages to eDoctor. One drawback of this approach is that if the developers implement their own mechanisms to manage preferences, eDoctor cannot track the changes. This is rare, however.

For system-wide configurations, eDoctor records changes that may affect battery usage, including changing CPU frequency, changing display brightness, changing display timeout, toggling Bluetooth connection, toggling GPS receiver, changing network type (2G/3G/4G), toggling Wi-Fi connection, toggling Airplane mode (which turns off wireless communications), toggling the background data setting, upgrading the system, and switching firmware. eDoctor records these events by capturing broadcast messages by the Android system. For example, when the Wi-Fi connection status changes, the system sends a broadcast message, `WIFI_STATE_CHANGED_ACTION`.

To protect user privacy, eDoctor stores the above information in its app-specific storage that other apps cannot access. In addition, it does not transfer the information outside of the phone; all analysis is done locally.

4.2 Data Analyzer

eDoctor's Data Analyzer is responsible for parsing all resource usage data collected by Information Collector, generating phase information (Section 3) for each app, and storing it in a per-app *phase table*. Since such phase analysis incurs overhead, it is only performed when the phone is being charged and the user is not interacting with the phone.

Every time when invoked, the Data Analyzer processes all the *analysis interval*s that haven't been analyzed. In our implementation, an analysis interval is one charging cycle, i.e., the time period between two phone charges. For each analysis interval, eDoctor identifies execution phases by using either RTV or RUV as explained in Section 3. To reduce noise and speed up diagnosis, it only records *major phases* – phases that account for more than 5% of the app's total execution time during the last analysis interval. Phases that appear occasionally are likely to be noise.

Each entry in a phase table represents a major phase. Each major phase is identified by a unique phase signature. We use phase signatures to determine which phase a given new resource vector belongs to. For RTV, we use the RTV vector directly as the phase signature; for RUV, we use the center and the radius of the corresponding cluster as the phase signature (refer to Section 3).

For each major phase, the Data Analyzer keeps track of its birth timestamp, and its number of appearances and energy consumption during each analysis interval. The birth timestamp helps diagnosis by indicating how recently a suspicious phase is first observed. The Diagnosis Engine also uses this information to correlate suspicious phases with triggering events (Section 4.3). For the last two variables (appearance count and energy consumed), only the most recent K intervals of data are maintained. Clearly, a large K allows for detection of issues that are introduced earlier, but it incurs larger storage and computing overhead and potential mis-diagnosis. We find $K = 7$ (about one week in time) strikes a good balance in the trade-off.

Figure 4 illustrates a simplified version of phase analysis. Based on k-means clustering computation (Section 3), entries with timestamp 5, 10 and 25 belong to the same phase (Phase #1 in the Phase Table below), because they have similar normalized usage patterns even though the absolute values of their entries differ largely. In addition, the entries at time 15 and 20 belong to the same phase (Phase #2), as the app only uses CPU for data processing (in this simplified example, we assume the values in the other columns for other resources are all zero). The entry at time 30 indicates that the app is not running, so it is not inserted in the Phase Table. The last entry at time 35 is another new phase (Phase #3) where only wakelock is held for a long time but the app does not use much other resources. It is the typical symptom when the developer forgets to release wakelock.

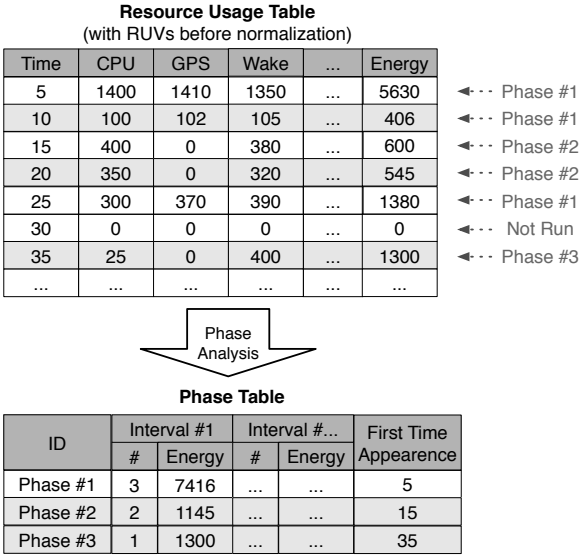


Figure 4: **Phase analysis illustration**. The resource usage table shows seven resource usage records collected by using the RUV method (before normalizing to CPU time).

4.3 Diagnosis Engine

When users notice ABD issues, they invoke eDoctor's Diagnosis Engine, which pinpoints the culprit app and the causing event. It analyzes historical phase tables (calculated by the Data Analyzer, Section 4.2) and event records (collected by the Information Collector, Section 4.1), and correlating them to identify the culprits.

Identifying the culprit app and the causing event is not trivial. As demonstrated in Section 3, energy profile and rank are not reliable indicators for diagnosing ABD issues. Some ABD issues are caused by intensive consumption of certain resources, e.g., GPS or the processor. However, the mere fact that a resource is used for a long time does not necessarily indicate abnormal behavior — an app may simply be designed to run for a long time. In addition, considering only recent resource usage is insufficient, since historical baseline data is needed to identify abnormal behavior.

eDoctor's approach is based on a key observation: most ABD issues involve a new, energy-heavy execution phase emerging in a particular app. For example, in the Facebook bug mentioned in Section 1, a new such phase is characterized by the wakelock being held for a long time while other resources are used little in the meantime. This phase rarely exhibited before the buggy upgrade. We refer to such a phase as a *suspicious new phase* (SN-Phase), and any app that contains an SN-Phase as a *suspicious app*. The diagnosis process has two major steps: (1) identifying suspicious apps, and then (2) identifying suspicious causing events.

Step 1: Identifying suspicious apps. eDoctor first prunes out apps that consume low energy, because they are unlikely the root cause of noticeable ABD. We only consider the top apps that, combined, consumed 90% of the energy. eDoctor then checks whether there is any recent SN-Phase. Determining whether a phase is energy-heavy or not is straightforward (e.g., by computing its energy consumption percentile in the app). But how to define *new*? Users may not start diagnosis immediately after an ABD issue happens. In other words, ABD may start well before the moment of diagnosis. In consideration of this, Diagnosis Engine uses a progressive strategy to search for suspicious apps as follows.

Recall that within an app, each major phase's information is recorded for the K most recent analysis intervals (i.e., charging cycles), which we notate as τ_1, τ_2, ..., τ_K, where τ_1 is the most recent interval and τ_K is the oldest interval. The Diagnosis Engine first assumes that the noticed ABD originally happened in τ_1. It thus treats those phases with birth timestamps falling in τ_2 to τ_K as normal ones where no ABD occurred. It then checks if τ_1 has any new energy-heavy phase appearing compared to the previous $K-1$ intervals. If it does not find any, it then assumes the ABD started in τ_2 (and may continue in τ_1), thus it checks whether any SN-Phase exists in the most recent two intervals, τ_1 and τ_2, compared to the previous $K-2$ intervals. The process goes on until it finally identifies an SN-Phase or it has exhausted all collected data in the phase table. For apps that are recently installed, they may not have much information in previous intervals. In such cases, any phase that consumes a high level of energy in recent analysis intervals is still considered to be an SN-Phase (when there is no previous intervals to compare). As mentioned before, all apps that contain SN-Phases are then regarded as suspicious apps. Based on our extensive empirical experiments (Section 5), there are usually at most 2–3 suspects after this step.

eDoctor keeps a week of historical data for each app, for two main reasons. First, if a new phase appears but the user has been using the app for a week without observing battery issues, that new phase is likely to be legitimate. Second, storing less data helps control the storage overhead and computation time of the data analysis (Section 4.2).

Step 2: Identifying suspicious causing events. For each suspicious app, the event that immediately precedes its SN-Phase is considered the most suspicious in causing the ABD. The Diagnosis Engine finds it by comparing the timestamp of the SN-Phase and the timestamps in the event logs.

Finally, the Diagnosis Engine ranks all suspicious apps based on the total energy consumed in their SN-Phase(s). For user convenience, eDoctor reports only the top ranked suspicious app and causing event for repair advisor. Certainly, it could also report all suspicious apps to experienced users if necessary.

4.4 Repair Advisor

In addition to providing a diagnosis report about the suspicious apps and causing events, eDoctor also suggests the most suitable repair solutions based on the symptom and causing events.

Uninstalling or reverting a problematic app to a previous version. If a recent update contains an ABD issue, eDoctor suggests to revert the problematic app back to the previous version or uninstall the app. Unfortunately, Android does not allow reverting apps directly. A tech-savvy user can revert an app with command line tools if a previous version is accessible. A better solution is to revert apps automatically by backing up prior installation packages. When Android installs an app, it stores the installation package on the phone temporarily, but it keeps only the last installed version of the package. If we back up prior versions, we can allow users to install prior versions. eDoctor has implemented a prototype and proved the feasibility of this approach.

Terminating apps after use. If the user wants to keep using the problematic version of the culprit app, eDoctor suggests temporary repair solutions in certain scenarios. One of the most common symptoms of energy bugs is that an app continues to consume resources even after the user stops using the app. In this case, eDoctor suggests users to manually terminate the problematic app every time after closing it, so it will not run in the background. As this can be troublesome, a better solution is to have eDoctor automatically terminate the problematic app.

Reverting configuration changes. If a recent configuration change causes an ABD issue, eDoctor presents users the identified configuration entry, together with its current and old values. It relies on the user to revert the configuration back to the old setting. User-level apps do not have permission to directly change configuration values. However, if implemented in the Android framework, it is possible to automatically fix configuration issues.

We leave the implementation and evaluation of automatic repair to future work.

5 Evaluation

To assess the effectiveness and performance overhead of eDoctor, we used real-world ABD issues to conduct both in-lab experiments and a controlled user study.

5.1 Effectiveness (User Study)

We wanted to evaluate eDoctor on real user phones, where ABD issues were mixed with normal usage of phones and apps. Thus, we recruited 31 Android users via campus-wide mailing lists in two major universities - University of California at San Diego (USA) and Peking University (China). These users had 26 different devices with 11 different Android versions and various configurations and usage patterns.

A real user study would ask participants to troubleshoot naturally occurring ABD issues. However, such a study might take several months and require a large number of participants to generate sufficient data points. Thus, we conducted a more controlled experiment. We emulated real-world scenarios where a user performed an ABD-triggering event (e.g., installing a buggy app or misconfiguring a setting), used the phone for some time, noticed rapid battery drain, and then started diagnosis. The whole study took 7–10 days for each participant.

ABD issues were hard to reproduce due to their dependency on specific versions of hardware and software. We finally reproduced 17 *real-world* ABD issues. We also generated 4 synthetic issues by modifying open-source Android apps (Table 3). We selected only popular apps that had a significant number of users; these apps also had heterogeneous patterns of user interactivity and resource utilization. Thus, we believe that our user study provides a relatively diverse and realistic sample.

For ABD issues caused by software bugs, we prepared two versions of a target app: one with a real-world ABD issue and the other without (i.e., either already fixed or not yet defective). We took similar steps with ABD-triggering configuration changes. Next, we randomly assigned each ABD issue from Table 3 to 1–5 participants, giving us 50 cases in total. In each case, we asked the user to follow three steps: (1) Use the given app (normal version) for at least 5 days. Meanwhile, participants should use their own apps as usual. (2) Switch the app to the defective version, or change the configuration to the incorrect one. To make it easy for participants to do this, we designed custom software that performed the switch with a single click. (3) Use the defective app until ABD is apparent, and then invoke eDoctor to diagnose the problem. In total, we collected 6,274 hours of real-world resource usage data. We used this data to evaluate eDoctor's diagnostic effectiveness, as well as its energy, storage, and memory overhead.

5.1.1 Diagnosis Result

Figure 5 shows eDoctor's effectiveness. Overall, eDoctor with RUV accurately diagnosed 47 of the 50 cases (94% accuracy). eDoctor misdiagnosed three cases. These three ABD issues were experienced by multiple users. eDoctor misdiagnosed these issues for some participants, but successfully diagnosed the issues for other users.

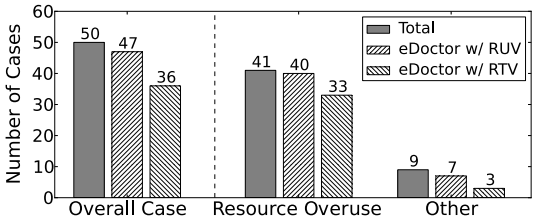


Figure 5: **Diagnosis results**. "Overall Case" shows the diagnosed cases among all 50 ABD cases. "Resource Overuse" and "Other" show breakdown of two types of ABD cases.

There were two reasons for misdiagnosis. First, some ABD issues occurred without an obvious change in the app's phase behavior. For example, at initialization time, one user configured the "Weather Bug" app to frequently update its weather data. High-frequency updates cause ABD, but for this user, the "Weather Bug" app *started* in the ABD state, so eDoctor could not detect anomalies in the app's behavior after the user upgraded to the defective version. eDoctor misdiagnosed ABD with the "K9Mail" app for a similar reason.

eDoctor can also misdiagnose ABD if it lacks sufficient longitudinal data for an ABD-causing app. For example, one user ran the non-buggy version of the "Vanilla

App Name	Category	Description	Downloads	Issue Type	Issue Description
Anki-Android	Education	A flash card app	100K+	Bug Config	Resource overuse (Accelerometer) Frequent widget refreshing
BostonBusMap	Travel	Bus tracking in Boston	50K+	Bug Config	Resource overuse (GPS) Enable continuous updates
Cool Reader*	Book	An eBook reader	1M+	Bug	Resource overuse (Wakelock)
Eyes-Free Shell	Tools	Eyes free access to apps	10K+	Bug	Resource overuse (GPS)
Facebook	Social	Official Facebook app	100M+	Bug	Resource overuse (Wakelock)
Gallery	Media	A 3D gallery app	built-in	Bug	Resource overuse (Accelerometer)
K9Mail	Communication	An popular email client	1M+	Bug	Too many trials
Marine Compass	Tools	A compass app	100K+	Bug	Resource overuse (Magnetic field sensor)
MyTracks	Health	Route tracking	5M+	Bug	Resource overuse (Wakelock and GPS)
Nice Compass	Tools	A compass app	1K+	Bug	Resource overuse (Magnetic field sensor)
NPR News*	News	NPR News client	1M+	Bug	Resource overuse (GPS)
OpenGPS Tracker	Travel	Route tracking	100K+	Bug Config	Resource overuse (GPS) GPS precision
OpenStreetMap	Productivity	OpenStreetMap viewer	5K+	Bug	Resource overuse (GPS)
Replica Island*	Game	An Android game	1M+	Bug	Resource overuse (Orientation sensor)
Standup Timer	Productivity	A timer app	1K+	Bug	Resource overuse (Orientation sensor)
Talking Dialer	Communication	A dialer app	50K+	Bug	Resource overuse (Accelerometer)
Vanilla*	Music	A music player	50K+	Bug	Resource overuse (Wakelock)
Weather Bug	Weather	A weather reporter	10M+	Config	Frequent update
WHERE	Travel	Location discovery	1M+	Bug	Resource overuse (GPS)

Table 3: **Apps and ABD issues used in our experiments**. The numbers in the "Downloads" column indicate the number of app downloads from Google Play, as of May 2012. To save space, we use "K" to present 1,000 and "M" for 1,000,000. "Built-in" means this app is bundled with some phones. To cover a wider spectrum of resources and usage patterns, we injected four real-world ABD bugs into apps in popular categories. They are marked with the "*" symbol. "Resource overuse" indicates a bug that uses a resource for longer than necessary, e.g., the developer forgets to release a resource after using it or holds a resource for too long.

Player" app for a short amount of time. During this short time period, the app displayed behaviors that resembled a wakelock leak (this might have occurred because the user frequently paused the player). The user soon updated to the defective version of the player that *did* have a wakelock leak. However, eDoctor did not detect a new phase, and thus could not flag the application as suspicious. If eDoctor is deployed to a large number of users, it can learn an apps phases using many different instances of that app. eDoctor could then leverage this large data set to identify even "early onset" ABD issues.

eDoctor is meant to be used as a diagnosis tool instead of a detection tool. When the user observes a battery drain and invokes eDoctor, it reports the app that is most likely to be the root cause. So we focused the evaluation on correct diagnosis vs. misdiagnosis instead of true positives vs. false positives.

RTV vs. RUV. As expected, RUV is more accurate than RTV; the former had an accuracy of 94%, but the latter only diagnosed 72% of cases correctly. RUV captures phase characteristics better than RTV, and can detect abnormal phases that use the same resources as their normal counterparts but in abnormal amounts. We also broke down the 50 cases into two high-level categories: resource overuse and other cases. RUV performs better than RTV in both categories. Interestingly, RTV is better at resource overuse (80.5%) than others (33.3%). The reason is that resource overuse often involve an app intensively using only one type of resource.

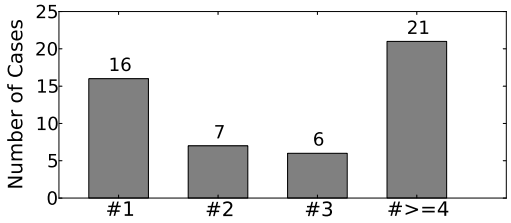
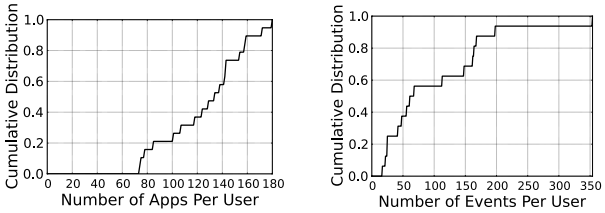


Figure 6: **Energy consumption rank of the culprit app**. The number at the top indicates the number of ABD cases, e.g., in 21 cases the rank is equal to or greater than 4.

Is the culprit app always the biggest energy consumer? As discussed in Section 1.2, one may wonder if existing energy profilers can detect ABD simply by identifying the top energy-consuming apps. Our data explains why this will not work. As illustrated in Figure 6, only 32% (16) of the cases have a culprit app that ranked #1 in energy use. In almost half (21) of the cases, the rank of the culprit app was greater than three. In these cases, the apps with ABD drained a significant amount of energy; however, other healthy, concurrently running apps also drew large amounts of energy (or the user noticed the ABD before the faulty app could waste a lot of energy). This demonstrates why existing profiling tools are insufficient for diagnosing many types of ABD. In addition, users may

be confused by the top-ranked but healthy apps and make wrong decisions to uninstall them or stop using them.

How many apps were monitored and how many events happened? As Figure 7 (a) shows, for at least 60% of the users, more than 120 apps are installed. The app counts include pre-installed apps and services that users were not even aware of. We also found that many energy-related events happened on the phone during the user study time period (7–10 days). As Figure 7 (b) shows, 60% of the users had at least 50 events taking place. As shown, eDoctor could diagnose culprits among all these events and monitored apps with high accuracy.



(a) (b)

Figure 7: **Distribution of the number of apps and events.**

5.1.2 Phase Distribution

To further understand the phase behavior of smartphone apps, we also examined how many normal phases smartphone apps may have. Figure 8 shows the cumulative distribution of all 1,890 apps that we monitored during the user study. The most important observation is that, during normal execution, most apps have a small number of major phases. For example, if using RTV (i.e., identifying phases based on resource type), about 80% of the apps have only 1 major phase in normal use and another 13% have only 2. If using RUV (i.e., considering resource usage amount), apps have more major phases, but 80% of the apps have at most 4 different phases. Section 3 described how eDoctor normalizes RUV with respect to CPU time. Figure 8 depicts the number of phases detected with and without normalization. As shown, normalization reduces the number of phases. After normalizing, nearly 75% of apps have only 1 normal phase.

5.2 Overhead

eDoctor's Information Collector periodically runs in the background (by default, once every 5 minutes). In this section, we describe eDoctor's overhead in terms of energy, storage, and memory.

Battery consumption overhead. We directly measured eDoctor's battery consumption on the Nexus One phone. We used a National Instruments NI USB-6210 DAQ to measure the voltage and current on the battery and calculate the power consumption of the entire device. As shown

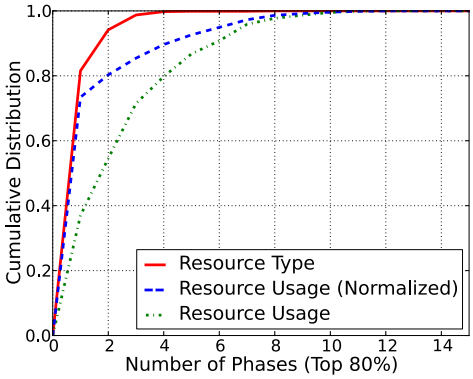


Figure 8: **The cumulative distribution of number of phases across 1,890 apps we monitored on real user phones during the user study**. We only consider the major phases that account for 80% of the total execution time.

in Figure 9, eDoctor added only 1.5% power overhead to an *idle* Nexus One (82.5mW) which had no user interaction but only ran built-in system software with Wi-Fi and radio signal enabled. eDoctor's energy overhead should be even lower—normal user activity will wake the phone up, allowing eDoctor to "piggyback" on this energy usage and collect resource statistics in the background. eDoctor's resource collection also has low overhead because eDoctor leverages Android's preexisting infrastructure for persistent resource tracking.

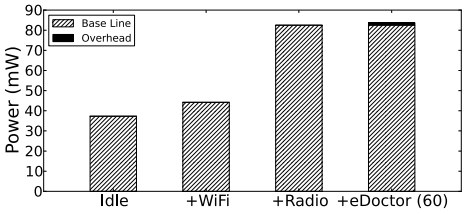


Figure 9: **eDoctor's battery consumption overhead for data collection**. Baseline (the first three bars): idle Nexus One phone with Wi-Fi and radio signal enabled. eDoctor collects all 60 active apps' resource usage on this phone (the fourth bar).

Storage overhead. eDoctor uses storage to collect resource utilization data and phase statistics. We measured this storage overhead by running a phone with eDoctor installed for 24 hours. eDoctor's overhead increases with the number of apps, so we ran experiments with 100, 125, and 150 installed apps. Table 4 shows that eDoctor consumed at most 3.2 MB per day. By default, eDoctor tracks one week of information; thus, eDoctor requires at most 22.4 MB of total storage. The storage overhead is even smaller in reality, because eDoctor does not store resource data if an app is not running. This is an acceptable overhead, since modern smartphones contain several gigabytes of storage space.

Number of Apps	100	125	150
Data size (24 hours)	1915 KB	2419 KB	2884 KB
Phase information	216 KB	270 KB	324 KB
Total	**2131 KB**	**2689 KB**	**3208 KB**

Table 4: **Storage used by eDoctor**

Memory Overhead. We used Trepn™ Profiler [3] to measure eDoctor's memory overhead. eDoctor's memory footprint was only 23.3 – 25.2MB. Memory utilization was stable over time because eDoctor only buffers a small amount of data and periodically stores the data to persistent storage.

6 Limitations and Discussions

When does eDoctor struggle? eDoctor has poor accuracy if the phases related to ABD were also common before the ABD began. This might happen if an application was initially started in a broken state (see Section 5.1.1). eDoctor's diagnostic accuracy will also suffer if a user simultaneously installs or reconfigures two apps, one of which is normal but energy-hungry, and another which has an ABD bug. In this scenario, eDoctor will regard both apps as suspect; since eDoctor only reports the highest ranked app, misdiagnosis may result. Such a situation did not arise during our user study, but finding automatic resolutions for such problems is an area for future work. Finally, misdiagnosis might also occur if the causal event occurred so long ago that it no longer resides in eDoctor's historical data.

Alternative approaches? While eDoctor leverages phase behavior to identify abnormal apps, but there are alternative approaches. For example, using signal processing techniques, one could detect abnormal energy consumption in the same way that network intrusion detectors identify traffic flows [9]. However, such techniques often have many false positives. One could also use dynamic bug detectors to identify code paths that may lead to ABD [22][33]. However, they introduce significant overhead because of the run-time instrumentation, which makes it hard to deploy directly to users' smartphones. In addition, they only work for ABD issues caused by already known bug patterns. In comparison, eDoctor is light weight and it can diagnose ABD issues caused by various types of misconfiguration, bugs and so on.

Is eDoctor limited to Android? Although we implemented eDoctor on Android, its approach is not limited to any particular platform. We chose Android because of its openness—we could record resource usage without users having to jailbreak their phones. We could also modify the platform to track app-specific configuration changes.

7 Related Work

Energy consumption modeling and measurement. Carroll et al. [13] measured power consumption of components in modern smartphones. Thiagarajan et al. [48] measured energy used by mobile browsers. Shye et al. [46] studied how user activities affect battery consumption, and derived a linear regression based power model. Zhang et al. [52] presented a power model based on system variables, e.g., the processor's frequency, the amount of data received through the network, and the display brightness. Recently, Pathak et al. [35, 36] proposed "Eprof", a tool that performs fine-grained energy profiling by tracing system calls. eDoctor leverages Android's internal energy usage tracker; this tracker has less overhead, but it is sufficiently accurate for eDoctor to effectively rank applications by their energy usage.

Malware detection by monitoring energy usage. Kim et al. [25] detects malware that causes sudden battery drain as a side-effect. Similar to malware detection for desktops/laptops, this work detects *"known"* battery-drain malware by comparing the power signature of each application in a smartphone with those known signatures stored in a malware database. eDoctor focuses on diagnosing battery-drain caused by *unknown* software bugs or configuration changes that may happen to any smartphone apps.

Energy-efficient smartphone design. Prior work covers a wide spectrum of system design: processors (GreenDroid [21]), resource management (ECOSystem [51], Cinder [42]), file systems (quFiles [49]), page allocation ([29]), I/O interfaces (Co-op I/O [50]), display ([5]), wireless networking (PGTP [4], STPM [6], SleepWell [32], SALSA [41], Bartendr [43]), and high level services (EnLoc [14], Micro-blog [20], EnTracked [26], A-loc [31]). These efforts focus on reducing energy usage in *normal* circumstances. In comparison, eDoctor troubleshoots abnormal battery drain.

Abnormal battery drain studies. Pathak et al. [34] conducted a study on battery issues on Android. Their results also show that ABD is an emerging problem. Recently, Pathak et al. [37] studied energy bugs in smartphone apps. They found many bugs are caused by failing to release resources and thus preventing a phone from switching to sleep mode (called "NoSleep" bugs). They also proposed a detector leveraging a reaching definition data flow analysis to detect the missed API calls that release resources. Their work can help developers to detect this specific type of energy bug in source code. eDoctor is complementary because (1) eDoctor helps *users* diagnose ABD issues and find the appropriate repairs; and (2) eDoctor does not assume that ABD is caused by specific types of bugs. Instead, eDoctor can diagnose ABD that arises from a variety of misconfigurations, bugs and so on.

8 Conclusions

This paper addresses the emerging abnormal battery drain (ABD) issue on smartphones. We built a practical tool, eDoctor, to help users diagnose and repair ABD issues. In our user study with 21 ABD issues and 31 participants, eDoctor successfully diagnosed 47 out of 50 cases with only small battery and storage overhead. We plan to release eDoctor on Google Play so that it can help real users while also collecting feedback for further improvement.

Acknowledgments

We greatly appreciate NSDI anonymous reviewers for their insightful feedback. We especially thank our shepherd, Dr. James W. Mickens, for his great effort to help us improve the paper. We also thank the members in the OPERA research group and the UCSD System and Networking (SysNet) group, and Erik Hinterbichler for their discussions and paper proof-reading. Last but not the least, we are grateful to the volunteers who participated our user study. This research is supported by NSF CNS–0720743 grant, NSF CSR Small 1017784 grant and NSF CSR–1217408 grant.

References

[1] Android 1.6 Platform Highlights - Battery Usage Indicator. http://developer.android.com/about/versions/android-1.6-highlights.html.

[2] App Store (iOS). http://en.wikipedia.org/wiki/App_Store_(iOS).

[3] Trepn ™Profiler. https://developer.qualcomm.com/mobile-development/development-devices/trepn-profiler.

[4] ANAND, B., SEBASTIAN, J., MING, S., ANANDA, A., CHAN, M., AND BALAN, R. Pgtp: Power Aware Game Transport Protocol for Multi-player Mobile Games. In *Proceedings of the International Conference on Communications and Signal Processing* (2011), ICCSP '11, pp. 399–404.

[5] ANAND, B., THIRUGNANAM, K., SEBASTIAN, J., KANNAN, P. G., ANANDA, A. L., CHAN, M. C., AND BALAN, R. K. Adaptive Display Power Management for Mobile Games. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (2011), MobiSys '11, ACM, pp. 57–70.

[6] ANAND, M., NIGHTINGALE, E. B., AND FLINN, J. Self-tuning Wireless Network Power Management. In *Proceedings of the 9th Annual International Conference on Mobile Computing and Networking* (2003), MobiCom '03, ACM, pp. 176–189.

[7] ANAND, M., NIGHTINGALE, E. B., AND FLINN, J. Ghosts in the Machine: Interfaces for Better Power Management. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services* (2004), MobiSys '04, ACM, pp. 23–35.

[8] BALASUBRAMANIAN, N., BALASUBRAMANIAN, A., AND VENKATARAMANI, A. Energy Consumption in Mobile Phones: a Measurement Study and Implications for Network Applications. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference* (2009), IMC '09, ACM, pp. 280–293.

[9] BARFORD, P., KLINE, J., PLONKA, D., AND RON, A. A Signal Analysis of Network Traffic Anomalies. In *Internet Measurement Workshop* (2002), pp. 71–82.

[10] BICKFORD, J., LAGAR-CAVILLA, H. A., VARSHAVSKY, A., GANAPATHY, V., AND IFTODE, L. Security Versus Energy Tradeoffs in Host-based Mobile Malware Detection. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (2011), MobiSys '11, ACM, pp. 225–238.

[11] CALIMLIM, A. Apple iOS 6.1 Reportedly Plagued With Battery, 3G and Syncing Issues. http://mashable.com/2013/02/09/ios-6-1-issues/, February 2013. by Mashable.

[12] CANALYS. Smartphones Overtake Client PCs in 2011. http://www.canalys.com/newsroom/smart-phones-overtake-client-pcs-2011, February 2012.

[13] CARROLL, A., AND HEISER, G. An Analysis of Power Consumption in a Smartphone. In *Proceedings of the USENIX Annual Technical Conference* (2010), USENIX ATC'10, USENIX Association, pp. 21–21.

[14] CONSTANDACHE, I., GAONKAR, S., SAYLER, M., CHOUDHURY, R., AND COX, L. EnLoc: Energy-efficient Localization for Mobile Phones. In *Proceedings of the 28th Conference on Computer Communications* (2009), INFOCOM '09, IEEE Computer Society, pp. 2716–2720.

[15] CUERVO, E., BALASUBRAMANIAN, A., CHO, D.-K., WOLMAN, A., SAROIU, S., CHANDRA, R., AND BAHL, P. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (2010), MobiSys '10, ACM, pp. 49–62.

[16] DHODAPKAR, A. S., AND SMITH, J. E. Comparing Program Phase Detection Techniques. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture* (2003), MICRO '36, IEEE Computer Society, pp. 217–228.

[17] DONG, M., AND ZHONG, L. Chameleon: A Color-adaptive Web Browser for Mobile OLED Displays. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (2011), MobiSys '11, ACM, pp. 85–98.

[18] DONG, M., AND ZHONG, L. Self-constructive High-rate System Energy Modeling for Battery-powered Mobile Systems. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (2011), MobiSys '11, ACM, pp. 335–348.

[19] DUESTERWALD, E., CASCAVAL, C., AND DWARKADAS, S. Characterizing and Predicting Program Behavior and Its Variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques* (2003), PACT '03, IEEE Computer Society, pp. 220–231.

[20] GAONKAR, S., LI, J., CHOUDHURY, R. R., COX, L., AND SCHMIDT, A. Micro-blog: Sharing and querying content through mobile phones and social participation. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services* (2008), MobiSys '08, ACM, pp. 174–186.

[21] GOULDING-HOTTA, N., SAMPSON, J., VENKATESH, G., GARCIA, S., AURICCHIO, J., HUANG, P.-C., ARORA, M., NATH, S., BHATT, V., BABB, J., SWANSON, S., AND TAYLOR, M. The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future. *IEEE Micro 31* (March 2011), 86–95.

[22] HANGAL, S., AND LAM, M. S. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of the 24th International Conference on Software Engineering* (2002), ICSE '02, ACM, pp. 291–301.

[23] HUANG, M. C., RENAU, J., AND TORRELLAS, J. Positional Adaptation of Processors: Application to Energy Reduction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture* (2003), ISCA '03, ACM, pp. 157–168.

[24] JUNE, L. Facebook Mobile App Stats Shocker: 104M iPhone Users, 12M Android Users.

http://www.engadget.com/2010/08/25/facebook-mobile-app-stats-shocker-104-million-iphone-users-12, August 2010. from engadget.

[25] KIM, H., SMITH, J., AND SHIN, K. G. Detecting Energy-greedy Anomalies and Mobile Malware Variants. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services* (2008), MobiSys '08, ACM, pp. 239–252.

[26] KJÆGAARD, M. B., LANGDAL, J., GODSK, T., AND TOFTKJÆR, T. EnTracked: Energy-efficient Robust Position Tracking for Mobile Devices. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services* (2009), MobiSys '09, ACM, pp. 221–234.

[27] LATEROID. JuiceDefender, an Battery Saving Application on Android. `http://latedroid.com/juicedefender`.

[28] LAU, J., PERELMAN, E., HAMERLY, G., SHERWOOD, T., AND CALDER, B. Motivation for Variable Length Intervals and Hierarchical Phase Behavior. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software* (2005), ISPASS '05, IEEE Computer Society, pp. 135–146.

[29] LEBECK, A. R., FAN, X., ZENG, H., AND ELLIS, C. Power Aware Page Allocation. *SIGARCH Comput. Archit. News 28*, 5 (Nov. 2000), 105–116.

[30] LIN, F. X., WANG, Z., LIKAMWA, R., AND ZHONG, L. Reflex: Using Low-power Processors in Smartphones Without Knowing Them. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS '12, ACM, pp. 13–24.

[31] LIN, K., KANSAL, A., LYMBEROPOULOS, D., AND ZHAO, F. Energy-accuracy Trade-off for Continuous Mobile Device Location. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (2010), MobiSys '10, ACM, pp. 285–298.

[32] MANWEILER, J., AND ROY CHOUDHURY, R. Avoiding the Rush Hours: WiFi Energy Management via Traffic Isolation. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (2011), MobiSys '11, ACM, pp. 253–266.

[33] NETHERCOTE, N., AND SEWARD, J. Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (2007), PLDI '07, ACM, pp. 89–100.

[34] PATHAK, A., HU, Y. C., AND ZHANG, M. Bootstrapping Energy Debugging on Smartphones: a First Look at Energy Bugs in Mobile Devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks* (2011), HotNets-X, ACM, pp. 5:1–5:6.

[35] PATHAK, A., HU, Y. C., AND ZHANG, M. Where is the Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones With Eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), pp. 29–42.

[36] PATHAK, A., HU, Y. C., ZHANG, M., BAHL, P., AND WANG, Y.-M. Fine-Grained Power Modeling for Smartphones Using System Call Tracing. In *Proceedings of the 6th ACM European Conference on Computer Systems* (2011), EuroSys '11, ACM, pp. 153–168.

[37] PATHAK, A., JINDAL, A., HU, Y. C., AND MIDKIFF, S. P. What is Keeping My Phone Awake?: Characterizing and Detecting No-sleep Energy Bugs in Smartphone Apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (2012), MobiSys '12, ACM, pp. 267–280.

[38] PERELMAN, E., HAMERLY, G., AND CALDER, B. Picking Statistically Valid and Early Simulation Points. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques* (2003), PACT '03, IEEE Computer Society, pp. 244–255.

[39] PERRUCCI, G., FITZEK, F., SASSO, G., KELLERER, W., AND WIDMER, J. On the Impact of 2G and 3G Network Usage for Mobile Phones' Battery Life. *Wireless Conference* (2009), 255–259.

[40] POWERS, R. Batteries for Low Power Electronics. *Proc. of the IEEE 83*, 4 (April 1995), 687–693.

[41] RA, M.-R., PAEK, J., SHARMA, A. B., GOVINDAN, R., KRIEGER, M. H., AND NEELY, M. J. Energy-delay Tradeoffs in Smartphone Applications. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (2010), MobiSys '10, ACM, pp. 255–270.

[42] ROY, A., RUMBLE, S. M., STUTSMAN, R., LEVIS, P., MAZIÈRES, D., AND ZELDOVICH, N. Energy Management in Mobile Devices with the Cinder Operating System. In *Proceedings of the 6th ACM European Conference on Computer Systems* (2011), EuroSys '11, ACM, pp. 139–152.

[43] SCHULMAN, A., NAVDA, V., RAMJEE, R., SPRING, N., DESHPANDE, P., GRUNEWALD, C., JAIN, K., AND PADMANABHAN, V. N. Bartendr: a Practical Approach to Energy-aware Cellular Data Scheduling. In *Proceedings of the 16th Annual International Conference on Mobile Computing and Networking* (2010), MobiCom '10, ACM, pp. 85–96.

[44] SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (2002), ASPLOS '02, ACM, pp. 45–57.

[45] SHERWOOD, T., SAIR, S., AND CALDER, B. Phase Tracking and Prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture* (2003), ISCA '03, ACM, pp. 336–349.

[46] SHYE, A., SCHOLBROCK, B., AND MEMIK, G. Into the Wild: Studying Real User Activity Patterns to Guide Power Optimizations for Mobile Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (2009), MICRO 42, ACM, pp. 168–178.

[47] SMITH, C. Google Play: 20 Billion App Downloads, 600,000 Apps and Games. http://www.androidauthority.com/google-play-20-billion-app-downloads-600000-apps-games-98006/, June 2012. by AndroidAuthority.

[48] THIAGARAJAN, N., AGGARWAL, G., NICOARA, A., BONEH, D., AND SINGH, J. P. Who Killed My Battery?: Analyzing Mobile Browser Energy Consumption. In *Proceedings of the 21st International Conference on World Wide Web* (2012), WWW '12, ACM, pp. 41–50.

[49] VEERARAGHAVAN, K., FLINN, J., NIGHTINGALE, E. B., AND NOBLE, B. quFiles: the Right File at the Right Time. *Trans. Storage 6* (September 2010), 12:1–12:28.

[50] WEISSEL, A., BEUTEL, B., AND BELLOSA, F. Cooperative I/O: a Novel I/O Semantics for Energy-aware Applications. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (2002), OSDI '02, ACM, pp. 117–129.

[51] ZENG, H., ELLIS, C. S., LEBECK, A. R., AND VAHDAT, A. ECOSystem: Managing Energy as a First Class Operating System Resource. *SIGOPS Oper. Syst. Rev. 36* (October 2002), 123–132.

[52] ZHANG, L., TIWANA, B., QIAN, Z., WANG, Z., DICK, R. P., MAO, Z. M., AND YANG, L. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *Proc. of the eighth IEEE/ACM/IFIP International Conference on Hardware/software Codesign and System Synthesis* (2010), CODES/ISSS '10, ACM, pp. 105–114.

ArrayTrack: A Fine-Grained Indoor Location System

Jie Xiong and Kyle Jamieson
University College London

Abstract

With myriad augmented reality, social networking, and retail shopping applications all on the horizon for the mobile handheld, a fast and accurate location technology will become key to a rich user experience. When roaming outdoors, users can usually count on a clear GPS signal for accurate location, but indoors, GPS often fades, and so up until recently, mobiles have had to rely mainly on rather coarse-grained signal strength readings. What has changed this status quo is the recent trend of dramatically increasing numbers of antennas at the indoor access point, mainly to bolster capacity and coverage with multiple-input, multiple-output (MIMO) techniques. We thus observe an opportunity to revisit the important problem of localization with a fresh perspective. This paper presents the design and experimental evaluation of ArrayTrack, an indoor location system that uses MIMO-based techniques to track wireless clients at a very fine granularity in real time, as they roam about a building. With a combination of FPGA and general purpose computing, we have built a prototype of the ArrayTrack system. Our results show that the techniques we propose can pinpoint 41 clients spread out over an indoor office environment to within 23 centimeters median accuracy, with the system incurring just 100 milliseconds latency, making for the first time ubiquitous real-time, fine-grained location available on the mobile handset.

1 Introduction

The proliferation of mobile computing devices continues, with handheld smartphones, tablets, and laptops a part of our everyday lives. Outdoors, these devices largely enjoy a robust and relatively accurate location service from Global Positioning System (GPS) satellite signals, but indoors where GPS signals don't reach, providing an accurate location service is quite challenging.

Furthermore, the demand for accurate location information is especially acute indoors. While the few meters of accuracy GPS provides outdoors are more than sufficient for street-level navigation, small differences in location have more importance to people and applications indoors: a few meters of error in estimated location can place someone in a different office within a building, for example. Location-aware smartphone applications on the horizon such as augmented reality-based building navigation, social networking, and retail shopping demand both a high accuracy and a low response time. A solution that offers a centimeter-accurate location service indoors would enable these and other exciting applications in mobile and pervasive computing.

Using radio frequency (RF) for location has many challenges. First, the many objects found indoors near access points (APs) and mobile clients reflect the energy of the wireless signal in a phenomenon called *multipath propagation*. This forces an unfortunate tradeoff that most existing RF location-based systems make: either model this hard-to-predict pattern of multipath fading, or leverage expensive hardware that can sample the wireless signal at a very high rate. Most existing RF systems choose the former, building maps of multipath signal strength [2, 3, 34, 43], or estimating coarse differences using RF propagation models [11, 14], achieving an average localization accuracy of between 60 cm [43] and meters: too coarse for the applications at hand.

Systems based on ultrasound and RF sensors such as Active Badge [35], Bat [36], and Cricket [19] have achieved accuracy to the level of centimeters, but usually require dedicated infrastructure to be installed in every room in a building, an approach that is expensive, time consuming, and requires maintenance effort.

Recently, however, two new opportunities have arisen in the design of indoor location systems:

1. WiFi APs are incorporating ever-increasing numbers of antennas to bolster capacity and coverage with multiple-input, multiple-output (MIMO) techniques. In fact, we expect that in the future, the number of antennas at the access point will increase several-fold, to meet the demand for MIMO links and spatial multiplexing [1, 31], which increase overall capacity. Indeed, the upcoming 802.11ac standard will specify eight MIMO spatial streams, and 16-antenna APs have been available since 2010 [41].
2. Meanwhile, WiFi AP density remains high: With our experimental infrastructure excluded, transmissions from most locations in our testbed reach seven or more production network APs, with all but about five percent of locations reaching five or more such APs. Furthermore, by leveraging the signal processing that is possible at the physical layer, an AP can extract information from a single packet at a lower SNR than

This material is based on work supported by the European Research Council under Grant No. 279976. Jie Xiong is supported by the Google European Doctoral Fellowship in Wireless Networking.

what is required to receive and decode the packet. This allows even more ArrayTrack APs to cooperate to localize clients than would be possible were the system to operate exclusively at higher layers.

ArrayTrack is an indoor localization system that exploits the increasing number of antennas at commodity APs to provide fine-grained location for mobile devices in an indoor setting. When a client transmits a frame on the air, multiple ArrayTrack APs overhear the transmission, and each compute angle-of-arrival (AoA) information from the clients' incoming frame. Then, the system aggregates the APs' AoA data at a central backend server to estimate the client's location. While AoA techniques are already in wide use in radar and acoustics, the challenge in realizing these techniques indoors is the presence of strong multipath RF propagation in these environments. To address this problem, we introduce novel system design techniques and signal processing algorithms that reliably eliminate the effects of multipath, even in the relatively common situations when little or no energy arrives on the direct path between client and AP.

ArrayTrack advances the state of the art in localization in three distinct ways:

1. To mitigate the effects of indoor multipath propagation, ArrayTrack contributes a novel *multipath suppression* algorithm to effectively remove the reflection paths between clients and APs.
2. Upon detecting a frame, an ArrayTrack AP quickly switches between sets of antennas, synthesizing new AoA information from each. We term this technique *diversity synthesis*, and find that it is especially useful in the case of low AP density.
3. ArrayTrack's system architecture centers around parallel processing in hardware, at APs, and in software, at the database backend, for fast location estimates.

We implement ArrayTrack on the Rice WARP FPGA platform and evaluate in a 41-node network deployed over one floor of a busy office space. Experimental results in this setting show that using just three APs, ArrayTrack can localize clients to a median 57 cm and mean one meter accuracy. With six APs, ArrayTrack achieves a median 23 cm and mean 31 cm location accuracy, localizing 95% of clients to within 90 cm. At the same time, ArrayTrack is fast, requiring just 100 milliseconds to produce a location estimate. To our knowledge, these are the most accurate and responsive location estimates available to date for an RF-based location system that does not require infrastructure except a normal density of nearby WiFi APs. Furthermore, as we experimentally show, ArrayTrack's performance is robust against different antenna heights, different antenna orientation, low SNR and collisions.

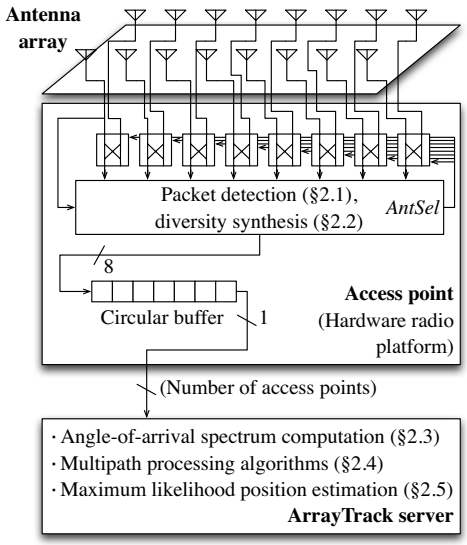


Figure 1: ArrayTrack's high-level design for eight radio front-ends, divided into functionality at each ArrayTrack access point and centralized server functionality. For clarity, we omit transmit path functionality of the access point.

In the next section, we detail ArrayTrack's design. An implementation discussion (§3) and performance evaluation (§4) then follow. We discuss related work in Section 5, and Section 7 concludes.

2 Design

We describe ArrayTrack's design as information flows through the system, from the physical antenna array, through the AP hardware, and on to the central ArrayTrack server, as summarized in Figure 1. First, ArrayTrack leverages techniques to detect packets at very low signal strength, so that many access points can overhear a single transmission (§2.1). Next, at each AP, ArrayTrack uses many antennas (§2.2) to generate an *AoA spectrum*: an estimate of likelihood versus bearing (§2.3), and cancels some of the effects of multipath propagation (§2.4). Finally, the system combines these estimates to estimate location (§2.5), further eliminating multipath's effects.

2.1 Packet detection and buffer management

To compute an AoA spectrum for a client, the AP only need overhear a small number of frames (between one and three, for reasons that will become clear in Section 2.4) from that client. For ArrayTrack's purposes, the contents of the frame are immaterial, so our system can process control frames such as acknowledgments, and even frames encrypted at the link layer.

The physical-layer preamble of an 801.11 frame contains known short and long training symbols, as shown in Figure 2. We use a modified version of Schmidl-Cox [25] detection to detect an incoming frame's short training symbols. Once the detection block senses a frame, it activates the diversity synthesis mechanism described in

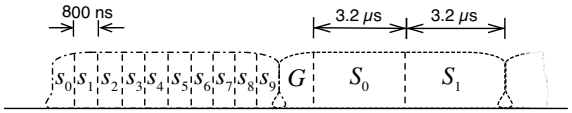


Figure 2: The 802.11 OFDM preamble consists of ten identical, repeated *short training symbols* (denoted $s_0 \ldots s_9$), followed by a *guard interval* (denoted G), ending with two identical, repeated *long training symbols* (denoted S_0 and S_1).

the next section and stores the samples of the incoming frame into a circular buffer, one logical buffer entry per frame detected.

Since it does not require even a partial packet decode, our system can process any part of the packet, which is helpful in the event of collisions in a carrier-sense multiple access network (§4.3.5). Our system detects the preamble of the packet and records a small part of it. In principle, one time domain packet sample would provide enough information for the AoA spectrum computation described in Section 2.3. However, to average out the effects of noise, we use 10 samples (we justify this choice in Section 4.3.3). Since a commodity WiFi AP samples at 40 Msamples/second, this implies that we need to process just 250 nanoseconds of a packet's samples, under 1.5% of an WiFi preamble's 16 μs duration.

2.2 Diversity synthesis

Upon detecting a packet, most commodity APs switch between pairs of antennas selecting the antenna from each pair with the strongest signal, a technique called *diversity selection*. This well-known and widely implemented technique improves performance in the presence of destructive multipath fading at one of the antennas, and can be found in the newest 802.11n MIMO access points today. It also has the advantage of not increasing the number of radios required, thus saving cost at the AP.

ArrayTrack seamlessly incorporates diversity selection into its design, synthesizing independent AoA data from both antennas of the diversity pair. We term this technique *diversity synthesis*.

Referring to Figure 1, once the packet detection block has indicated the start of a packet, control logic stores the samples corresponding to the preamble's long training symbol S_0 (Figure 2) from the upper set of antennas into the first half of a circular buffer entry. Next, the control logic toggles the *AntSel* (for antenna select) line in Figure 1, switching to the lower set of antennas for the duration of the second long training symbol S_1.[1] Since S_1 and S_2 are identical and each 3.2 μs long, they fall well within the coherence time[2] of the indoor wireless channel, and so we can treat the information obtained from each set of antennas as if the two sets were obtained simultaneously from different radios at the AP.

[1] We use the long training symbols because our hardware radio platform has a 500 ns switching time during which the received signal is highly distorted and unusable.

[2] The time span over which the channel can be considered stationary. Coherence time is mainly a function of the RF carrier frequency and speed of motion of the transmitter, receiver, and any nearby objects.

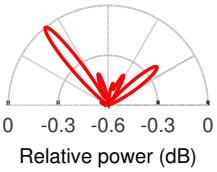


Figure 3: The *AoA spectrum* of a client's received signal at a multi-antenna access point estimates the incoming signal's power as a function of its angle of arrival.

2.3 AoA spectrum generation

Especially in indoor wireless channels, RF signals reflect off objects in the environment, resulting in multiple copies of the signal arriving at the access point: this phenomenon is known as multipath propagation. An AoA spectrum of a client's received signal at a multi-antenna AP is an estimate of the incoming signal's power as a function of angle of arrival, as shown in Figure 3. Since strong multipath propagation is present indoors, the direct-path signal may be significantly weaker than the reflected-path signals, or may even be undetectable. In these situations, the highest peak on the AoA spectrum would correspond to a reflected path instead of the direct path to the client. This makes indoor localization using AoA spectra alone highly inaccurate, necessitating the remaining steps in ArrayTrack's processing chain.

2.3.1 Phased-array primer

In order to explain how we generate AoA spectra, we now present a brief primer on phased arrays. While their technology is well established, we focus on indoor applications, highlighting the resulting complexities.

For clarity of exposition, we first describe how an AP can compute angle of arrival information in free space (*i.e.*, in the absence of multipath reflections), and then generalize the principles to handle multipath wireless propagation. The key to computing a wireless signal's angle of arrival is to analyze received *phase* at the AP, a quantity that progresses linearly from zero to 2π every RF wavelength λ along the path from client to access point, as shown in Figure 4 (*left*).

This means that when the client sends a signal, the AP receives it with a phase determined by the path length d from the client. Phase is particularly easy to measure at the physical layer, because software-defined and hardware radios represent the phase of the wireless signal graphically using an *in-phase-quadrature* (I-Q) plot, as shown in Figure 4 (*right*), where angle measured from the I axis indicates phase. Using the I-Q plot, we see that a distance d adds a phase of $2\pi d/\lambda$ as shown by the angle

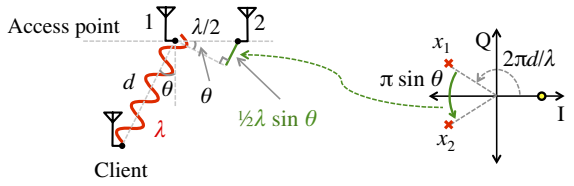


Figure 4: Principle of operation for ArrayTrack's AoA spectrum computation phase. (*Left:*) The phase of the signal goes through a 2π cycle every radio wavelength λ, and the distance differential between the client and successive antennas on the access point is related to the client's bearing on the access point. (*Right:*) The complex representation of the sent signal at the client (filled dot) and received signals at the access point (crosses) reflects this relationship.

measured from the I axis to the cross labeled x_1 (representing the signal received at antenna one). Since there is a $\lambda/2$ separation between the two antennas, the distance along a path arriving at bearing θ is a fraction of a wavelength greater to the second antenna than it is to the first, that fraction depending on θ. Assuming $d \gg \lambda/2$, the added distance is $\frac{1}{2}\lambda \sin\theta$.

These facts suggest a particularly simple way to compute θ at a two-antenna access point in the absence of multipath: measure x_1 and x_2 directly, compute the phase of each ($\angle x_1$ and $\angle x_2$), then solve for θ as

$$\theta = \arcsin\left(\frac{\angle x_2 - \angle x_1}{\pi}\right) \tag{1}$$

Generalizing to multiple antennas. In indoor multipath environments, Equation 1 quickly breaks down, because multiple paths' signals sum in the I-Q plot. However, adding multiple, say M, antennas can help. The best known algorithms are based on eigenstructure analysis of an $M \times M$ *correlation matrix* $\mathbf{R_{xx}}$ in which the entry at the l^{th} column and m^{th} row is the mean correlation between the l^{th} and m^{th} antennas' signals.

Suppose D signals $s_1(t), \ldots, s_D(t)$ arrive from bearings $\theta_1, \ldots, \theta_D$ at $M > D$ antennas, and that $x_j(t)$ is the received signal at j^{th} antenna element at time t. Recalling the relationship between measured phase differences and AP bearing discussed above, we use the *array steering vector* $\mathbf{a}(\theta)$ to characterize how much added phase (relative to the first antenna) we see at each of the other antennas, as a function of the incoming signal's bearing. For a linear array,

$$\mathbf{a}(\theta) = \exp\left(\frac{-j2\pi d}{\lambda}\right)\begin{bmatrix} 1 \\ \exp(-j\pi\lambda\cos\theta) \\ \exp(-j2\pi\lambda\cos\theta) \\ \vdots \\ \exp(-j(M-1)\pi\lambda\cos\theta) \end{bmatrix} \tag{2}$$

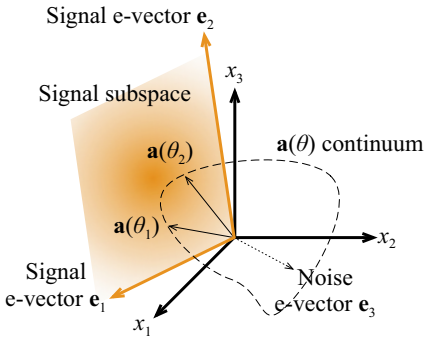


Figure 5: In this three-antenna example, the two incoming signals (at bearings θ_1 and θ_2 respectively) lie in a three-dimensional space. Eigenvector analysis identifies the two-dimensional *signal subspace* shown, and MUSIC traces along the array steering vector continuum measuring the distance to the signal subspace. Figure adapted from Schmidt [26].

So we can express what the AP hears as

$$\mathbf{x}(t) = \overbrace{[\mathbf{a}(\theta_1)\ \mathbf{a}(\theta_2)\ \cdots\ \mathbf{a}(\theta_D)]}^{\mathbf{A}}\begin{bmatrix} s_1(t) \\ s_2(t) \\ \vdots \\ s_D(t) \end{bmatrix} + n(k), \tag{3}$$

where $\mathbf{n}(k)$ is noise with zero mean and σ_n^2 variance. This means that we can express $\mathbf{R_{xx}}$ as

$$\begin{aligned} \mathbf{R_{xx}} &= \mathbb{E}[\mathbf{x}\mathbf{x}^*] \\ &= \mathbb{E}\left[(\mathbf{A}\mathbf{s} + \mathbf{n})\left(\mathbf{s}^*\mathbf{A}^* + \mathbf{n}^*\right)\right] \\ &= \mathbf{A}\mathbb{E}\left[\mathbf{s}\mathbf{s}^*\right]\mathbf{A}^* + \mathbb{E}\left[\mathbf{n}\mathbf{n}^*\right] \\ &= \mathbf{A}\mathbf{R_{ss}}\mathbf{A}^* + \sigma_n^2\mathbf{I} \end{aligned} \tag{4}$$

where $\mathbf{R_{ss}} = \mathbb{E}\left[\mathbf{s}\mathbf{s}^*\right]$ is the source correlation matrix.

The array correlation matrix $\mathbf{R_{xx}}$ has M eigenvalues $\lambda_1, \ldots, \lambda_M$ associated respectively with M eigenvectors $\mathbf{E} = [\mathbf{e}_1\ \mathbf{e}_2\ \cdots\ \mathbf{e}_M]$. If the noise is weaker than the incoming signals, then when the eigenvalues are sorted in non-decreasing order, the smallest $M - D$ correspond to the noise while the next D correspond to the D incoming signals. The D value depends on how many eigenvalues are considered big enough to be signals. We choose D value as how many eigenvalues are larger than a threshold that is a fraction of the largest eigenvalue. Based on this process, the corresponding eigenvectors in $\mathbf{E}$ can be classified as noise or signal:

$$\mathbf{E} = \left[\overbrace{e_1\ \ldots\ e_{M-D}}^{\mathbf{E}_N}\ \overbrace{e_{M-D+1}\ \ldots\ e_M}^{\mathbf{E}_S}\right] \tag{5}$$

we refer to $\mathbf{E}_N$ as the *noise subspace* and $\mathbf{E}_S$ as the *signal subspace*.

The MUSIC AoA spectrum [26] inverts the distance between a point moving along the array steering vector

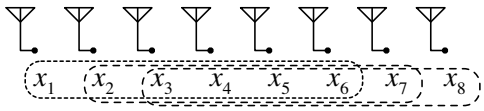


Figure 6: Spatial smoothing an eight-antenna array $x_1, \ldots, x_8$ to a virtual six-element array (number of groups $N_G = 3$).

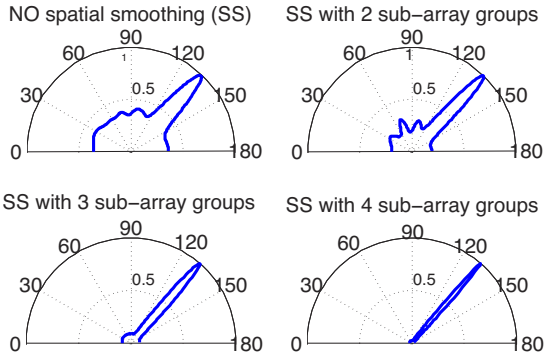


Figure 7: Varying the amount of spatial smoothing on AoA spectra.

continuum and $\mathbf{E}_S$, as shown in Figure 5:

$$P(\theta) = \frac{1}{\mathbf{a}(\theta)\mathbf{E}_N\mathbf{E}_N^*\mathbf{a}(\theta)} \tag{6}$$

This yields sharp peaks in $P(\theta)$ at the signals' AoA.

2.3.2 *Spatial smoothing for multipath distortion*

Implementing MUSIC as-is, however, yields highly distorted AoA spectra, for the following reason. When the incoming signals are phase-synchronized with each other (as results from multipath) MUSIC perceives the distinct incoming signals as one superposed signal, resulting in false peaks in $P(\theta)$. We therefore adopt *spatial smoothing* [28], averaging incoming signals across N_G groups of antennas to reduce this correlation. For example, spatial smoothing over $N_G = 3$ six-antenna groups on an eight-antenna array generating signals $x_1, \ldots, x_8$ would output six signals $\dot{x}_1, \ldots, \dot{x}_6$ where $\dot{x}_i = \frac{1}{3}(x_i + x_{i+1} + x_{i+2})$ for $1 \le i \le 6$, as shown in Figure 6.

How should we choose N_G? Figure 7 shows a microbenchmark computing MUSIC AoA spectra for a client near and in the line of sight of the AP (so the direct-path bearing dominates $P(\theta)$) both with and without spatial smoothing. As N_G increases, the effective number of antennas decreases, and so spatial smoothing can eliminate smaller peaks that may correspond to the direct path. On the other hand, as N_G increases, the overall noise in the AoA spectrum decreases, and some peaks may be narrowed, possibly increasing accuracy. Based on this microbenchmark and experience generating AoA spectra indoors from a number of different clients in our testbed, we find that a good compromise is to set $N_G = 2$, and we use this in the performance evaluation in Section 4.

Scenario	Frequency
Direct path same; reflection paths changed	71%
Direct path same; reflection paths same	18%
Direct path changed; reflection paths changed	8%
Direct path changed; reflection paths same	3%

Table 1: Peak stability microbenchmark measuring the frequency of the direct and reflection-path peaks changing due to slight movement.

2.3.3 *Array geometry weighting*

Information from the linear array we use in our system is not equally reliable as a function of θ, because of the asymmetric physical geometry of the array. Consequently, after computing a spatially-smoothed MUSIC AoA spectrum, the ArrayTrack multiplies it by a *windowing function* $W(\theta)$, the purpose of which is to weight information from the AoA spectrum in proportion to the confidence that we have in the data. With a linear array, we multiply $P(\theta)$ by

$$W(\theta) = \begin{cases} 1, & \text{if } 15^\circ < |\theta| < 165^\circ \\ \sin\theta, & \text{otherwise.} \end{cases} \tag{7}$$

2.3.4 *Array symmetry removal*

Although a linear antenna array can determine bearing, it cannot determine which side of the array the signal is arriving from. This means that the AoA spectrum is essentially a 180° spectrum mirrored to 360°. When there are many APs cooperating to determine location, this is not a problem, but when there are few APs, accuracy suffers. To address this, we employ the diversity synthesis scheme described in Section 2.2 to have a ninth antenna not in the same row as the other eight included. Using the ninth antenna, we calculate the total power on each side, and remove the half with less power, resulting in a true 360° AoA spectrum.

2.4 Multipath suppression

While the spatial smoothing algorithms described above (§2.3.2) reduce multipath-induced distortion of the AoA spectrum to yield an accurate spectrum, they don't identify the direct path, leaving multipath reflections free to reduce system accuracy. The multipath suppression algorithm we present here has the goal of removing or reducing peaks in the AoA spectrum not associated with the direct path from AP to client.

ArrayTrack's multipath suppression algorithm leverages changes in the wireless channel that occur when the transmitter or objections in the vicinity move by grouping together AoA spectra from multiple frames, if available. Our method is motivated by the following observation: when there are small movements of the transmitter, the receiver, or objects between the two, the direct-path peak on the AoA spectrum is usually stable while the reflection-path peaks usually change significantly, and

Algorithm (Multipath suppression)

1. Group two to three AoA spectra from frames spaced closer than 100 ms in time; if no such grouping exists for a spectrum, then output that spectrum to the synthesis step (§2.5).
2. Arbitrarily choose one AoA spectrum as the *primary*, and remove peaks from the primary not paired with peaks on other AoA spectra.
3. Output the primary to the synthesis step (§2.5).

Figure 8: ArrayTrack's multipath suppression algorithm.

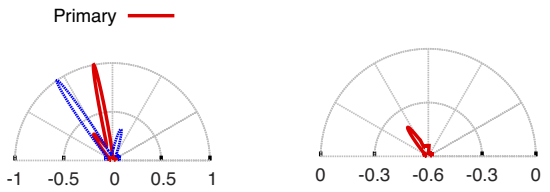


Figure 9: ArrayTrack's multipath suppression algorithm operating on two example AoA spectra (*left*) and the output AoA spectrum (*right*).

these slight movements happen frequently in real life when we hold a mobile handset making calls, for example.

We run a microbenchmark at 100 randomly chosen locations in our testbed (see Figure 12, p. 7), generating AoA spectra at each position selected and another position five centimeters away. If the corresponding bearing peaks of the two spectra are within five degrees, we mark that bearing as *unchanged*. If the variation is more than five degrees or the peak vanishes, we mark it *changed*.

The results are shown in Table 1. Most of the time, the direct-path peak is unchanged while the reflection-path peaks are changed. This motivates the algorithm shown in Figure 8. Note that for those scenarios in which both the direct-path and reflection-path peaks are unchanged, we keep all of them without any deleterious consequences. Also, observe that the microbenchmark above only captures two packets. This leaves room for even further improvement if we capture multiple packets during the course of the mobile's movement. The only scenario which induces failure in the multipath suppression algorithm is when the reflection-path peaks remain unchanged while the direct-path peak is changed. However, as shown above, the chances of this happening are small. We show and example of the algorithm's operation in Figure 9.

2.5 AoA spectra synthesis

In this step, ArrayTrack combines the AoA spectra of several APs into a final location estimate. Suppose N APs generate AoA spectra $P_1(\theta), \ldots, P_N(\theta)$ as processed by the previous steps, and we wish to compute the likelihood of the client being located at position $\mathbf{x}$ as shown in Figure 3. ArrayTrack computes the bearing of $\mathbf{x}$ to AP i, θ_i, by trigonometry, and then estimates the likelihood of the client being at location $\mathbf{x}$, $L(\mathbf{x})$, as

$$L(\mathbf{x}) = \prod_{i=1}^{N} P_i(\theta_i). \quad (8)$$

With Equation 8 we search for the most likely location of the client by forming a 10 centimeter by 10 centimeter grid, and evaluating $L(\mathbf{x})$ at each point in the grid. We then use hill climbing on the three positions with highest $L(\mathbf{x})$ in the grid using the gradient defined by Equation 8 to refine our location estimate.

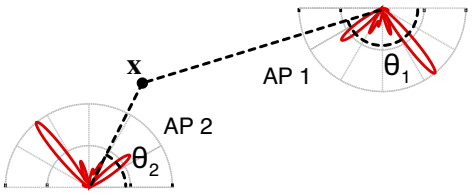


Figure 10: ArrayTrack combines information from multiple APs into a likelihood of the client being at location $\mathbf{x}$ by considering all AoA spectra at their respective bearings (θ_1, θ_2) to $\mathbf{x}$.

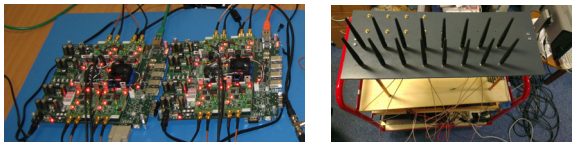


Figure 11: *Left:* the ArrayTrack prototype AP is composed of two WARP radios, while a cable-connected USRP2 software-defined radio (not shown) calibrates the array. *Right:* The AP mounted on a cart, showing its antenna array.

3 Implementation

The prototype ArrayTrack AP, shown in Figure 11, uses two Rice WARP FPGA-based wireless radios. Each WARP is equipped with four radio front ends and four omnidirectional antennas. We utilize the digital I/O pins on one of the WARP boards to output a time synchronization pulse on a wire connected between the two WARPs, so that the second WARP board can record and buffer the same time-indexed samples as the first. The WARPs run a custom FPGA hardware design architected with Xilinx System Generator for DSP that implements all the functionality described in Section 2.

We place the 16 antennas[3] attached to the WARP radios in a rectangular geometry (Figure 11, right). Antennas are spaced at a half wavelength distance (6.13 cm) to yield maximum AoA spectrum resolution. This also happens to yield maximum MIMO wireless capacity, and so is the arrangement preferred in commodity APs.

AP phase calibration. Equipping the AP with multiple antennas is necessary for ArrayTrack, but does not

[3]The two WAPPs have a total of eight radio boards, each with two ports. ArrayTrack is able to switch ports as described in §2.2 and record the two long training symbols with different antennas. So with two WARPs, the maximum number of antennas we can utilize is 16.

suffice to calculate angle of arrival as described in the preceding section. Each radio receiver incorporates a 2.4 GHz oscillator whose purpose is to convert the incoming radio frequency signal to its representation in I-Q space shown, for example, in Figure 4 (p. 4). An undesirable consequence of this *downconversion* step is that it introduces an unknown phase offset to the resulting signal, rendering AoA inoperable. This is permissible for MIMO, but not for our application, because this manifests as an unknown phase added to the constellation points in Figure 4. Our solution is to calibrate the array with a USRP2 generating a continuous wave tone, measuring each phase offset directly. Because small manufacturing imperfections exist for SMA splitters and cables labelled the same length, we propose a one-time (run only once for a particular set of hardware) calibration scheme to handle these equipment imperfections.

The signal from the USRP2 goes through splitters and cables (we called them external paths) before reaching WARP radios. The phase offset Ph_{off} we want to measure is the internal phase difference $Ph_{in2} - Ph_{in1}$. Running calibration once, we obtain the following offset:

$$Ph_{off1} = (Ph_{ex2} + Ph_{in2}) - (Ph_{ex1} + Ph_{in1}) \quad (9)$$

Because of equipment imperfections, Ph_{ex2} is slightly different from Ph_{ex1} so Ph_{off1} is not equal to Ph_{off}. We exchange the external paths and run calibration again:

$$Ph_{off2} = (Ph_{ex1} + Ph_{in2}) - (Ph_{ex2} + Ph_{in1}) \quad (10)$$

Combing the above two equations, we obtain Ph_{off} and the phase difference caused by equipment imperfections:

$$(Ph_{off2} + Ph_{off1})/2 = Ph_{off} \quad (11)$$

$$(Ph_{off2} - Ph_{off1})/2 = Ph_{ex1} - Ph_{ex2} \quad (12)$$

Subtracting the measured phase offsets from the incoming signals over the air then cancels the unknown phase difference, and AoA becomes possible.

Testbed clients. The clients we use in our experiments are Soekris boxes equipped with Atheros 802.11g radios operating in the 2.4 GHz band.

4 Evaluation

To show how well ArrayTrack performs in real indoor environment, we present experimental results from the testbed described in Section 3. First we present the accuracy level ArrayTrack achieves in the challenging indoor office environment and explore the effects of number of antennas and number of APs on the performance of ArrayTrack. After that, we demonstrate that ArrayTrack is robust against different transmitter/receiver heights and different antenna orientations between clients and APs. Finally we examine the latency introduced by ArrayTrack, which is a critical factor for a real-time system.

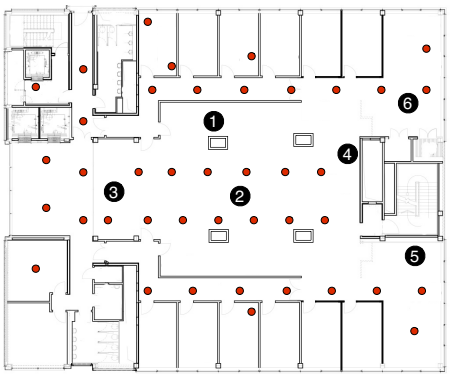


Figure 12: Testbed environment: Soekris clients are marked as small dots, and the AP locations are labelled "1"–"6".

Experimental methodology. We place prototype APs at the locations marked "1"–"6" in our testbed floorplan, shown in Figure 12. The layout shows the basic structure of the office but does not include the numerous cubicle walls also present. We place the 41 clients roughly uniformly over the floorplan, covering areas both near to, and far away from the AP. We put some clients near metal, wood, glass and plastic walls to make our experiments more comprehensive. We also place some clients behind concrete pillars in our office so that the direct path between the AP and client is blocked, making the situation more challenging.

To measure ground truth in the location experiments presented in this section, we used scaled architectural drawings of our building combined with measurements taken from a Fluke 416D laser distance measurement device, which has an accuracy of 1.5 mm over 60 m.

Due to budget constraints, we used one WARP AP, moving it between the different locations marked on the map in Figure 12 and receiving new packets to emulate many APs receiving a transmission simultaneously. This setup does not favor our evaluation of ArrayTrack.

4.1 Static localization accuracy

We first evaluate how accurately AoA pseudospectrum computation without array geometry weighting and reflection path removing localizes clients. This represents the performance ArrayTrack would obtain in a static environment without any client movement, or movement nearby. The curves labeled three APs, four APs, five APs, and six APs in Figure 13 show raw location error computed with Equation 8 across all different AP combinations and all 41 clients. We see that the general trend is that average error decreases with an increasing number of APs. The median error varies from 75 cm for three APs to 26 cm for six APs. The average error varies from 317 cm for three APs to 38 cm for six APs. We show a heatmap combination example in Figure 14 with increasing number of APs.

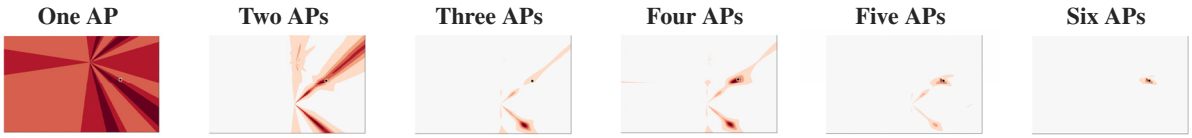


Figure 14: Heatmaps showing the location likelihood of a client with differing numbers of APs computing its location. We denote the ground truth location of the client in each by a small dot in each heatmap.

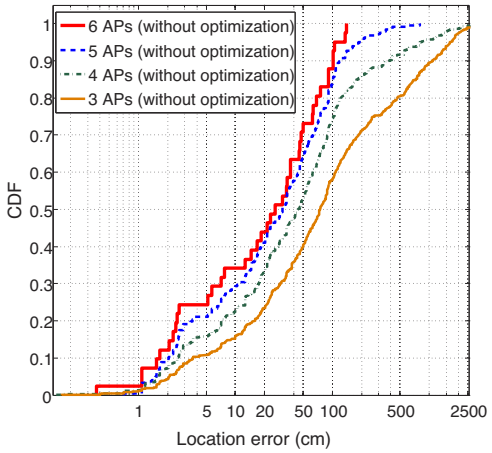


Figure 13: Cumulative distribution of location error from unoptimized raw AoA spectra data across clients using measurements taken at all combinations of three, four, five, and six APs.

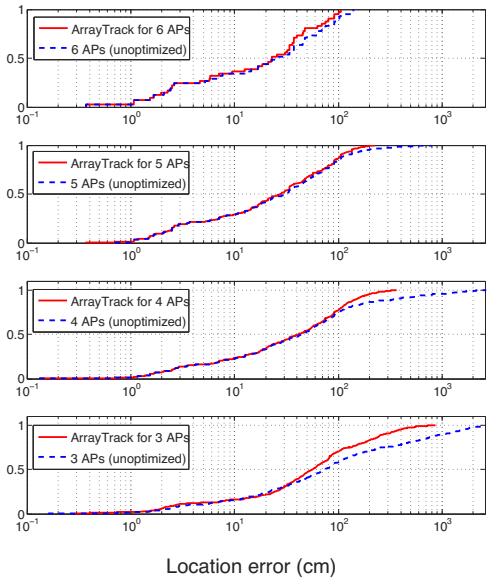


Figure 15: Cumulative distribution of location error across clients for three, four, five and six APs with ArrayTrack.

4.2 Semi-static localization accuracy

We now evaluate ArrayTrack using data that incorporates small (less than 5 cm) movements of the clients, with two more such location samples per client. This is representative of human movement even when stationary, due to small inadvertent movements, and covers all cases where there is even more movement up to walking speed. In Figure 15, we show that ArrayTrack improves the accuracy level greatly, especially when the number of APs is small. Our system improves mean accuracy level from 38 cm to 31 cm for six APs (a 20% improvement). We measure 90%, 95% and 98% of clients to be within 80 cm, 90 cm and 102 cm respectively of their actual positions. This improvement is mainly due to the array geometry weighting, which removes the relatively inaccurate parts of the spectrum approaching 0 degrees or 180 degree (close to the line of the antenna array).

When there are only three APs, ArrayTrack improves the mean accuracy level from 317 cm to 107 cm, which is around a 200% improvement. The intuition behind this large performance improvement is the effective removal of the false positive locations caused by multipath reflections and redundant symmetrical bearings. When the number of APs is big such as five or six, heatmap combination inherently reinforces the true location and removes false positive locations. However, when the number of APs is small, this reinforcement is not always strong and sometimes the array symmetry causes false positive locations, which greatly degrades the localization performance. In these cases, we enable the array symmetry removal scheme described in Section 2.3.4 to significantly enhance accuracy. By using this technique, ArrayTrack can achieve a median 57 cm accuracy levels with only three APs, good enough for many indoor applications.

4.2.1 Varying number of AP antennas

We now show how ArrayTrack performs with differing number of antennas at APs. In general, with more antennas at each AP, we can achieve a more accurate AoA spectrum and capture a higher number of reflection-path bearings, which accordingly increases localization accuracy, as Figure 16 shows. Because we apply spatial smoothing on top of the MUSIC algorithm, the effective number of antennas is actually reduced and so we are not able to capture all the arriving signals when the number of antennas is small. The mean accuracy level is 138 cm for four antennas, 60 cm for six antennas and 31 cm for eight antennas. It's interesting to note that the improvement gap between four and six antennas is bigger than that between six to eight antennas. In a strong multipath indoor environment like our office, the direct path signal is not always the strongest. However, the direct path signal is among the three biggest signals most of the time. We show how the direct path peak changes in Figure 17.

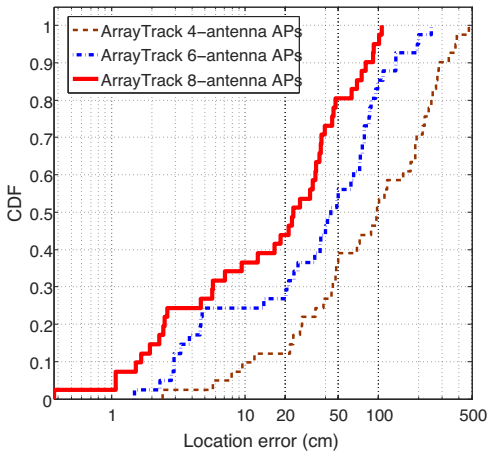


Figure 16: CDF plot of location error for four, six and eight antennas with ArrayTrack.

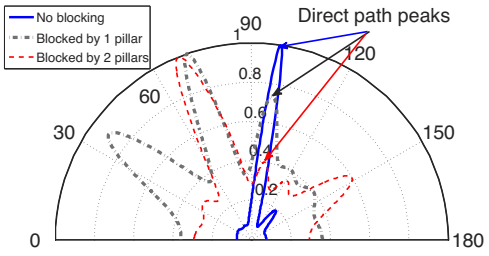


Figure 17: The AoA spectra for 3 clients in a line with AP.

We keep the client on the some line with AP while block it with more pillars. Even when it's blocked by two pillars, the direct path signal is still among the top three biggest, although not the strongest. With five virtual antennas, after spatial smoothing, we are able to avoid losing the direct path signals as sometimes happens when we only use four antennas. The accuracy level improvement from six to eight antennas is due to the more accurate AoA spectrum obtained. With an increasing number of antennas, there will be some point when increasing the number of antennas does not improve accuracy any more as the dominant factor will be the calibration, antenna imperfection, noise, correct alignment of antennas, and even the human measurement errors introduced with laser meters in the experiments. We expect that an antenna array with six antennas (30.5 cm long) or eight antennas (43 cm long) is quite reasonable.

4.3 Robustness

Robustness to varying client height, orientation, low SNR, and collisions is an important characteristic for ArrayTrack to achieve. We investigate ArrayTrack's accuracy under these adverse conditions in this section.

As ArrayTrack works works on any part of the packet. We choose the preamble of the packet to work with ArrayTrack. Preamble part is transmitted at the base rate and what's more, complex conjugate with the known

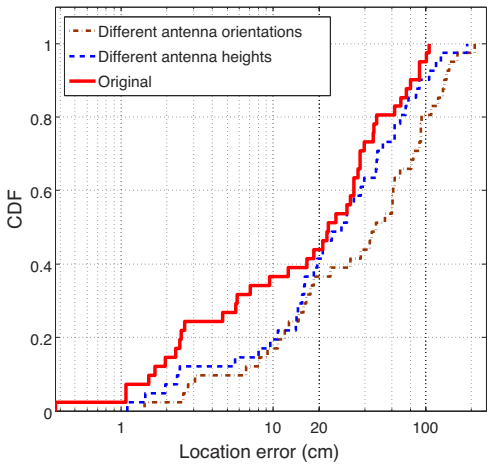


Figure 18: CDF plot of ArrayTrack's location error for different antenna height, different orientation and baseline results, with eight antennas and six APs.

training symbol generate peaks which is very easy to be detected even at low SNR.

4.3.1 *Height of mobile clients*

In reality, most mobiles rest on a table or are held in the hand, so they are often located around 1–1.5 meters off the ground. APs are usually located on the wall near the ceiling, typically 2.5 to 3 meters high. We seek to study whether this height difference between clients and APs will cause significant errors in our system's accuracy. The mathematical analysis in §2.3 is based on the assumption that clients and APs are at the same height. In Appendix A we show that a 1.5 meter height difference introduces just 1%–4% error when the distance between the client and AP varies between five and 10 meters. In our experiments, our AP is placed on top of a cart for easy movement with the antennas positioned 1.5 meters above the floor. To emulate a 1.5-meter height difference between AP and clients, we put the clients on the ground at exactly the same location and generate the localization errors with ArrayTrack to compare with the results obtained when they are more or less on the same height with the AP.[4]

The experimental results shown in Figure 18 demonstrate the preceding. Median location error is slightly increased from 23 cm to 26 cm when the AP uses eight antennas. One factor involved is that it is unlikely for a client to be close to all APs, as the APs are separated in space rather than being placed close to each other. One advantage of our system is the independence of each AP from the others, *i.e.*, when we have multiple APs, even if one of them is generating inaccurate results, the rest will not be affected and will mitigate the negative effects of the inaccurate AP by reinforcing the correct location.

[4]Note that this low height does not favor our experimental results as lower AP positions are susceptible to even more clutter from objects than an AP mounted high on the well near the ceiling.

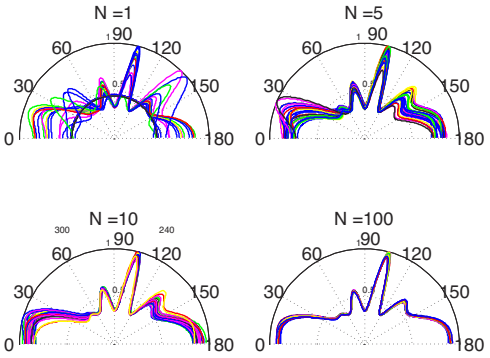


Figure 19: The effect of number of data samples on AoA spectrum.

In future work, we are planning to extend the ArrayTrack system to three dimensions by using a vertically-oriented antenna array in conjunction with the exiting horizontally-oriented array. This will allow the system to estimate elevation directly, and largely avoid this source of error entirely.

4.3.2 *Mobile orientation*

Users carry mobile phones in their hands at constantly-changing orientations, so we study the effect of different antenna orientations on ArrayTrack. Keeping the transmission power the same on the client side, we rotate the clients' antenna orientations perpendicular to the APs' antennas. The results in Figure 18 show that the accuracy level we achieve suffers slightly compared with the original results, median location error increasing from 23 cm to 50 cm. By way of explanation, we find that the received power at the APs is smaller with the changed antenna orientation, because of the different polarization. With linearly polarized antennas, a misalignment of polarization of 45 degrees will degrade the signal up to 3 dB and a misaligned of 90 degrees causes an attenuation of 20 dB or more. By using circularly-polarized antennas at the AP, this issue can be mitigated.

4.3.3 *Number of preamble samples*

To show that ArrayTrack works well with very small number of samples, we present testbed results in Figure 19. Each subplot is composed of 30 AoA spectra from 30 different packets recorded from the same client in a short period of time. We use different number of samples to generate our AoA pseudospectra. As WARPLab samples 40 MHz per second, one sample takes only 0.025 us. We can see that when the number of samples increased to 5, the AoA spectrum is already quite stable which demonstrate ArrayTrack has the potential to responds extremely fast. We employ 10 samples in our experiments and for a 100 ms refreshing interval, the overhead introduced by ArrayTrack traffic is as little as: $\frac{(10\text{ samples})(32\text{ bits/sample})(8\text{ radios})}{100\text{ ms}} = 0.0256$ Mbps.

4.3.4 *Low signal to noise ratio (SNR)*

We show the signal to noise ratio (SNR) effect on the performance of ArrayTrack in this section. Because ArrayTrack does not need to decode any packet content, all the short and long training symbols can be used for packets detection, which performs very well compared with the original Schmidl-Cox packet detection algorithm. With all the 10 short training symbols used, we are able to detect packets at SNR as low as -10 dB.

It's clear that low SNR is not affecting our packet detection at all. Then we want to see whether this low SNR affect our AoA performance. We keep the client at the same position untouched and keep decreasing the transmission power of the client to see how AoA spectra change. The results are shown in Figure 4.3.4. It can be seen clearly that when the SNR becomes very low below 0 dB, the spectrum is not sharp any more and very large side lobe appears on the spectrum generated. This will definitely affect our localization performance. However, we also find that as long as the SNR is not below 0 dB, ArrayTrack works pretty well.

4.3.5 *Packet collisions*

When there are two simultaneous transmissions which causes collision, ArrayTrack still works well as long as the preambles of the two packets are not overlapping.

For collision between two packets of 1000 bytes each, the chance of preamble colliding is 0.6%. We show that as long as the training symbols are not overlapping, we are able to obtain AoA information for both of them using a form of successive interference cancellation. We detect the first colliding packet and generate an AoA spectrum. Then we detect the second colliding packet and generate its AoA spectrum. However, the second AoA spectrum is composed of bearing information for both packets. So we remove the AoA peaks of the first packet from the second AoA spectrum, thus successfully obtaining the AoA information for the second packet.

4.4 System latency

System latency is important for the real-time applications we envision ArrayTrack will enable, such as augmented reality. Figure 21 summarizes the latency our system incurs, starting from the beginning of a frame's preamble as it is received by the ArrayTrack APs. As discussed previously (§4.3.3), ArrayTrack only requires 10 samples from the preamble in order to function. We therefore have the opportunity to begin transferring and processing the AoA information while the remainder of the preamble and the body of the packet is still on the air, as shown in the figure. System latency is comprised of the following pieces:

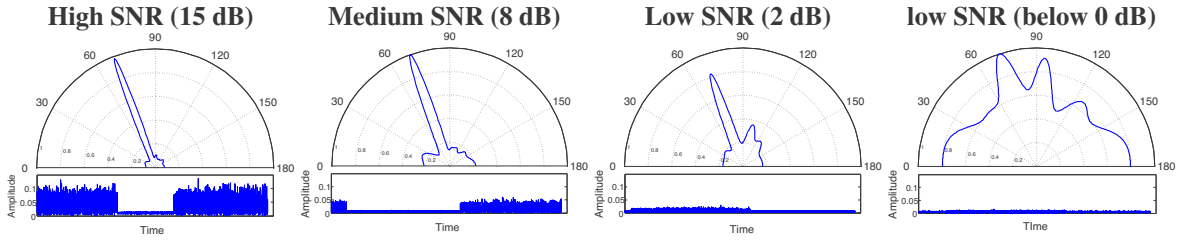


Figure 20: AoA spectra become less sharp and more side peaks when the SNR becomes small.

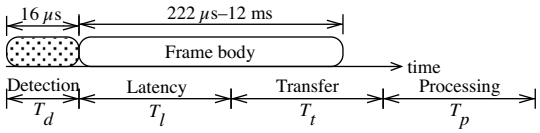


Figure 21: A summary of the end-to-end latency that the ArrayTrack system incurs in determining location.

1. T: the air time of a frame. This varies between approximately 222 μs for a 1500 byte frame at 54 Mbit/s to 12 ms for the same size frame at 1 Mbit/s.

2. T_d: the preamble detection time. For the 10 short and two long training symbols in the preamble, this is 16 μs.

3. T_l: WARP-PC latency to transfer samples. We estimate this to be approximately 30 milliseconds, noting that this can be significantly reduced with better bus connectivity such as PCI Express on platforms such as the Sora [32].

4. T_t: WARP-PC serialization time to transfer samples.

5. T_p, the time to process all recorded samples.

T_t is determined by the number of samples transferred from the WARPs to the PC and the transmission speed of the Ethernet connection. The Ethernet link speed between the WARP and PC is 100 Mbit/s. However, due to the very simple IP stack currently implemented on WARP, added overheads mean that the maximum throughput that can be achieved is about 1 Mbit/s. This yields $T_t = \frac{(10 \text{ samples})(32 \text{ bits/sample})(8 \text{ radios})}{1 \text{ Mbit/s}} = 2.56$ ms.

T_p depends on how the MUSIC algorithm is implemented and the computational capability of the ArrayTrack server. For an eight-antenna array, the MUSIC algorithm involves eigenvalue decomposition and matrix multiplications of linear dimension eight. Because of the small size of these matrices, this process is not the limiting factor in the server-side computations. In the synthesis step (§2.5) we apply a hill climbing algorithm to find the maximum in the heatmap computed from the AoA spectra. For our current Matlab implementation with an Intel Xeon 2.80 GHz CPU and 4 GB of RAM, the average processing time is 100 ms with a variance of 3 ms for the synthesis step.

Therefore, the total latency that ArrayTrack adds starting from the end of the packet (excluding bus latency) is $T_l = T_d + T_t + T_p - T \approx 100$ ms.

5 Related Work

The present paper is based on the ideas sketched in a previous workshop paper [39], but contributes novel diversity synthesis (§2.2) and multipath suppression (§2.4) design techniques and algorithms, as well as providing the first full performance evaluation of our system.

ArrayTrack owes its research vision to early indoor location service systems that propose dedicated infrastructure Active Badge [35] equips mobiles with infrared transmitters and buildings with many infrared receivers. The Bat System [36] uses a matrix of RF-ultrasound receivers, each hard-coded with location, deployed on the ceiling indoors. Cricket [19] equips buildings with combined RF/ultrasound beacons while mobiles carry RF/ultrasound receivers.

Some recent work including CSITE [13] and Pin-Loc [27] has explored using the OFDM subcarrier channel measurements as unique signatures for security and localization. This requires a large amount of wardriving, and the accuracy is limited to around one meter, while ArrayTrack achieves finer accuracy and eliminates any calibration beforehand.

The most widely used RF information is received signal strength (RSS), usually measured in units of whole decibels. While readily available from commodity WiFi hardware at this granularity, the resulting RSS measurements are very coarse compared to direct physical-layer samples, and so incur an amount of quantization error, especially when few readings are present.

Map-building approaches. There are two main lines of work using RSS; the first, pioneered by RADAR [2, 3] builds "maps" of signal strength to one or more access points, achieving an accuracy on the order of meters [23, 30]. Later systems such as Horus [43] use probabilistic techniques to improve localization accuracy to an average of 0.6 meters when an average of six access points are within range of every location in the wireless LAN coverage area, but require large amounts of calibration. While some work has attempted to reduce the calibration overhead [12], mapping generally requires significant calibration effort. Other map-based work has

proposed using overheard GSM signals from nearby towers [34], or dense deployments of desktop clients [4]. Recently, Zee [21] has proposed using crowd-sourced measurements in order to perform the calibration step, resulting in an end-to-end median localization error of three meters when Zee's crowd-sourced data is fed into Horus. In contrast to these map-based techniques, ArrayTrack achieves better accuracy with few APs, and requires no calibration of any kind beforehand, essential if there are not enough people nearby to crowd-source measurements before the RF environment changes.

Model-based approaches. The second line of work using RSS are techniques based on mathematical models. Some of these proposals use RF propagation models [22] to predict distance away from an access point based on signal strength readings. By triangulating and extrapolating using signal strength models, TIX [11] achieves an accuracy of 5.4 meters indoors. Lim *et al.* [14] use a singular value decomposition method combined with RF propagation models to create a signal strength map (overlapping with map-based approaches). They achieve a localization error of about three meters indoors. EZ [8] is a system that uses sporadic GPS fixes on mobiles to bootstrap the localization of many clients indoors. EZ solves these constraints using a genetic algorithm, resulting in a median localization error of between 2–7 meters indoors, without the need for calibration.

Other model-based proposals augment RF propagation models with Bayesian probabilistic models to capture the relationships between different nodes in the network [16], and develop conditions for a set of nodes to be localizable [42]. Still other model-based proposals are targeted towards ad hoc mesh networks [6, 20, 24].

Prior work in AoA. Wong *et al.* [37] investigate the use of AoA and channel impulse response measurements for localization. While they have demonstrated positive results at a very high SNR (60 dB), typical wireless LANs operate at significantly lower SNRs, and the authors stop short of describing a complete system design of how the ideas would integrate with a functioning wireless LAN as ArrayTrack does. Niculescu *et al.* [17] simulate AoA-based localization in an ad hoc mesh network. AoA has also been proposed in CDMA mobile cellular systems [40], in particular as a hybrid approach between TDoA and AoA [9, 38], and also in concert with interference cancellation and ToA [33].

Much other work in AoA uses the technology to solve similar but materially different problems. Geo-fencing [29] utilizes directional antennas and a frame coding approach to control APs' indoor coverage boundary. Patwari *et al.* [18] propose a system that uses the channel impulse response and channel estimates of probe tones to detect when a device has moved, but do not address location. Faria and Cheriton [10] and others [5, 15] have proposed using AoA for location-based security and behavioral fingerprinting in wireless networks. Chen *et al.* [7] investigate *post hoc* calibration for commercial off-the-shelf antenna arrays to enable AoA determination, but do not investigate localization indoors.

6 Discussion

How does ArrayTrack deal with NLOS?
The NLOS encountered in our experiments can be categorized into two different scenarios:

- S1: Direct path signal is not the strongest but exists.
- S2: Direct path signal is totally blocked.

S1 does not affect ArrayTrack as the spectra synthesis method strengthens the true location in nature.

For S2, one blocked direct path degrades the performance of ArrayTrack slightly but not much. It's not very likely the client's direct paths to all the APs are blocked.

Linear versus circular array arrangement?
Most commonly seen commercial APs have their antennas placed in linear arrangement. As circular array resolves 360 degrees while linear resolves 180 degrees, twice the number of antennas is needed for circular array to achieve the same level of resolution accuracy while linear array has the problem of symmetry ambiguity addressed with synthesis of multiple APs. We plan to consider other array arrangements in our future work.

7 Conclusion

We have presented ArrayTrack, an indoor location system that uses angle-of-arrival techniques to locate wireless clients indoors to a level of accuracy previously only attainable with expensive dedicated hardware infrastructure. ArrayTrack combines best of breed algorithms for AoA based direction estimation and spatial smoothing with novel algorithms for suppressing the non-line of sight reflections that occur frequently indoors and synthesizing information from many antennas at the AP.

A AP-Client Height Difference

Suppose the AP is distance h above the client; we compute the resulting percentage error. AoA relies on the distance difference $d_1 - d_2$ between the client and the two AP antennas in a pair. Given an added height, this difference becomes:

$$d_1' - d_2' = \frac{d_1}{\cos\phi} - \frac{d_2}{\cos\phi} \tag{13}$$

where $\cos\phi = h/d$. The percentage error is then $\frac{(d_1'-d_2')-(d_1-d_2)}{d_1-d_2} = (\cos\phi)^{-1} - 1$. For $h = 1.5$ meters and $d = 5$ meters, this is 4% error; for $h = 1.5$ meters and $d = 10$ meters, this is 1% error.

References

[1] E. Aryafar, N. Anand, T. Salonidis, and E. Knightly. Design and experimental evaluation of multi-user beamforming in wireless LANs. In *Proc. of ACM MobiCom*, 2010.

[2] P. Bahl and V. Padmanabhan. RADAR: An in-building RF-based user location and tracking system. In *Proc. of IEEE Infocom*, pages 775–784, 2000.

[3] P. Bahl, V. Padmanabhan, and A. Balachandran. Enhancements to the RADAR location tracking system. Technical Report MSR-TR-2000-12, Microsoft Research, Feb. 2000.

[4] P. Bahl, J. Padhye, L. Ravindranath, M. Singh, A. Wolman, and B. Zill. DAIR: A framework for managing enterprise wireless networks using desktop infrastructure. In *Proc. of ACM HotNets*, 2005.

[5] S. Bratus, C. Cornelius, D. Kotz, and D. Peebles. Active behavioral fingerprinting of wireless devices. In *Proc. of ACM WiSec*, pages 56–61, Mar. 2008.

[6] S. Capkun, M. Hamdi, and J. Hubaux. GPS-free positioning in mobile ad-hoc networks. In *Proc. of Hawaii Int'l Conference on System Sciences*, 2001.

[7] H. Chen, T. Lin, H. Kung, and Y. Gwon. Determining RF angle of arrival using COTS antenna arrays: a field evaluation. In *Proc. of the MILCOM Conf.*, 2012.

[8] K. Chintalapudi, A. Iyer, and V. Padmanabhan. Indoor localization without the pain. In *Proc. of ACM MobiCom*, 2010.

[9] L. Cong and W. Zhuang. Hybrid TDoA/AoA mobile user location for wideband CDMA cellular systems. *IEEE Trans. on Wireless Communications*, 1 (3):439–447, 2002.

[10] D. Faria and D. Cheriton. No long-term secrets: Location based security in overprovisioned wireless lans. In *Proc. of the ACM HotNets Workshop*, 2004.

[11] Y. Gwon and R. Jain. Error characteristics and calibration-free techniques for wireless LAN-based location estimation. In *ACM MobiWac*, 2004.

[12] A. Haeberlen, E. Flannery, A. Ladd, A. Rudys, D. Wallach, and L. Kavraki. Practical robust localization over large-scale 802.11 wireless networks. In *Proc. of ACM MobiCom*, 2004.

[13] Z. Jiang, J. Zhao, X. Li, J. Han, and W. Xi. Rejecting the Attack: Source Authentication for Wi-Fi Management Frames using CSI Information. In *Proc. of IEEE Infocom*, 2013.

[14] H. Lim, C. Kung, J. Hou, and H. Luo. Zero configuration robust indoor localization: Theory and experimentation. In *Proc. of IEEE Infocom*, 2006.

[15] D. C. Loh, C. Y. Cho, C. P. Tan, and R. S. Lee. Identifying unique devices through wireless fingerprinting. In *Proc. of the ACM WiSec Conf.*, pages 46–55, Mar. 2008.

[16] D. Madigan, E. Einahrawy, R. Martin, W. Ju, P. Krishnan, and A. Krishnakumar. Bayesian indoor positioning systems. In *Proc. of IEEE Infocom*, 2005.

[17] D. Niculescu and B. Nath. Ad-hoc positioning system (APS) using AoA. In *Proc. of IEEE Infocom*, 2003.

[18] N. Patwari and S. Kasera. Robust location distinction using temporal link signatures. In *Proc. of the ACM MobiCom Conf.*, pages 111–122, Sept. 2007.

[19] N. Priyantha, A. Chakraborty, and H. Balakrishnan. The Cricket location-support system. In *Proc. of the ACM MobiCom Conf.*, pages 32–43, Aug. 2000.

[20] N. Priyantha, H. Balakrishnan, E. Demaine, and S. Teller. Mobile-assisted localization in wireless sensor networks. In *Proc. of IEEE Infocom*, 2005.

[21] A. Rai, K. Chintalapudi, V. Padmanabhan, and R. Sen. Zee: Zero-effort crowdsourcing for indoor localization. In *Proc. of ACM MobiCom*, 2012.

[22] T. S. Rappaport. *Wireless Communications: Principles and Practice*. Prentice-Hall, 2nd edition, 2002.

[23] T. Roos, P. Myllymaki, and H. Tirri. A probabilistic approach to WLAN user location estimation. *International J. of Wireless Information Networks*, 9(3), 2002.

[24] A. Savvides, C. Han, and M. Srivastava. Fine-grained localization in ad-hoc networks of sensors. In *Proc. of ACM MobiCom*, 2001.

[25] T. Schmidl and D. Cox. Robust Frequency and Timing Synchroniation for OFDM. *IEEE Trans. on Communications*, 45(12):1613–1621, Dec. 1997.

[26] R. Schmidt. Multiple emitter location and signal parameter estimation. IEEE *Trans. on Antennas and Propagation*, AP-34(3):276–280, Mar. 1986.

[27] S. Sen, B. Radunovic, R. Choudhury, and T. Minka. Spot localization using phy layer information. In *Proceedings of ACM MOBISYS*, 2012.

[28] T.-J. Shan, M. Wax, and T. Kailath. On spatial smoothing for direction-of-arrival estimation of coherent signals. *IEEE Trans. on Acoustics, Speech, and Sig. Proc.*, ASSP-33(4):806–811, Aug. 1985.

[29] A. Sheth, S. Seshan, and D. Wetherall. Geofencing: Confining Wi-Fi Coverage to Physical Boundaries. In *Proceedings of the 7th International Conference on Pervasive Computing*, 2009.

[30] A. Smailagic, D. Siewiorek, J. Anhalt, D. Kogan, and Y. Wang. Location sensing and privacy in a context aware computing environment. In *Pervasive Computing*, 2001.

[31] K. Tan, H. Liu, J. Fang, W. Wang, J. Zhang,

M. Chen, and G. Voelker. SAM: Enabling practical spatial multiple access in wireless LAN. In *Proc. of ACM MobiCom*, 2009.

[32] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G. Voelker. Sora: High performance software radio using general purpose multi-core processors. In *Proc. of the* NSDI *Conf.*, Apr. 2009.

[33] A. Tarighat, N. Khajehnouri, and A. Sayed. Improved wireless location accuracy using antenna arrays and interference cancellation. 4, 2003.

[34] A. Varshavsky, E. Lara, J. Hightower, A. LaMarca, and V. Otsason. GSM indoor localization. In *Pervasive and Mobile Computing*, 2007.

[35] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The active badge location system. *ACM Trans. on Information Systems*, 10(1):91–102, Jan. 1992.

[36] A. Ward, A. Jones, and A. Hopper. A new location technique for the active office. *IEEE Personal Communications*, 4(5):42–47, Oct. 1997.

[37] C. Wong, R. Klukas, and G. Messier. Using WLAN infrastructure for angle-of-arrival indoor user location. In *Proc. of the IEEE VTC Conf.*, pages 1–5, Sept. 2008.

[38] Y. Xie, Y. Wang, P. Zhu, and X. You. Grid-search-based hybrd ToA/AoA location techniques for NLOS environments. *IEEE Comms. Letters*, 13 (4):254–256, 2009.

[39] J. Xiong and K. Jamieson. Towards fine-grained radio-based indoor location. In *Proc. of ACM HotMobile*, 2012.

[40] L. Xiong. A selective model to suppress nlos signals in angle-of-arrival (AoA) location esimation. In *Proc. of the IEEE PIMRC*, 1998.

[41] xirrus. Xirrus Corp. (http://www.xirrus.com).

[42] Z. Yang, Y. Liu, and X. Li. Beyond trilateration: On the localizability of wireless ad-hoc networks. In *Proc. of IEEE Infocom*, 2009.

[43] M. Youssef and A. Agrawala. The Horus WLAN location determination system. In *Proc. of ACM MobiSys*, 2005.

Walkie-Markie: Indoor Pathway Mapping Made Easy

Guobin Shen,[†] Zhuo Chen,[‡] Peichao Zhang,[‡] Thomas Moscibroda,[†] Yongguang Zhang[†]
Microsoft Research Asia, Beijing, China
[†]{jackysh, moscitho, ygz}@microsoft.com, [‡]{czxxdd, starforever00}@gmail.com

Abstract

We present Walkie-Markie – an indoor pathway mapping system that can automatically reconstruct internal pathway maps of buildings without any a-priori knowledge about the building, such as the floor plan or access point locations. Central to Walkie-Markie is a novel exploitation of the WiFi infrastructure to define landmarks (WiFi-Marks) to fuse crowdsourced user trajectories obtained from inertial sensors on users' mobile phones. WiFi-Marks are special pathway locations at which the trend of the received WiFi signal strength changes from increasing to decreasing when moving along the pathway. By embedding these WiFi-Marks in a 2D plane using a newly devised algorithm and connecting them with calibrated user trajectories, Walkie-Markie is able to infer pathway maps with high accuracy. Our experiments demonstrate that Walkie-Markie is able to reconstruct a high-quality pathway map for a real office-building floor after only 5-6 rounds of walks, with accuracy gradually improving as more user data becomes available. The maximum discrepancy between the inferred pathway map and the real one is within 3m and 2.8m for the anchor nodes and path segments, respectively.

1 Introduction

Accurate and inexpensive indoor localization is one of the holy grails of mobile computing, as it is the key to enabling indoor location-based services. Despite very significant research effort, relatively little has actually been deployed at scale. One reason is that a common and critical assumption of existing approaches – the availability of a suitable localization map – is hard to fulfill in practice. For instance, WiFi triangulation or fingerprinting based approaches for indoor localization rely on a priori AP position information, or a signal strength map to function properly [4, 13, 24]. Such maps are typically constructed via dedicated, often labor-intensive, data-gathering processes that map radio signals onto an indoor map that *geographically reflects the physical layout* of the building. Several recent efforts aimed at alleviating the pain of radio map construction require knowledge of the real floor plans [26, 38]. Similarly, tracking based localization also requires accurate indoor maps (e.g., floor plans or pathway maps) to constrain the drifting of inertia sensors [36, 37]. Such indoor maps are difficult to obtain in general, as they may belong to different owners, may be outdated, and many legacy buildings simply do not have them at all.

In this paper, we try to fundamentally rethink the assumption and ask the question: can we build an indoor map *without any prior knowledge* about the building? In particular, we are interested in building *pathway* maps because they provide a natural framework for localizing users and points of interest (POIs) as people usually move along pathways and indoor POIs are connected via pathways. A pathway map can also serve as the basis for other maps specific to other localization approaches, or can be used as a building block to construct semantically richer maps for users, for example through automatic location detection (e.g., [3]) or crowdsourced user annotation. Finally, we seek a technology that allows to obtain such pathway maps *at scale*, say for millions of buildings across the world, including shopping malls and office buildings.

We address these problems in *Walkie-Markie*, a system that automatically generates indoor pathway maps from traces contributed by mobile phone users. The system uses crowdsourcing to generate the pathway map of unknown buildings without requiring any a-priori information such as floor-plans, any initial measurements or inspection, and any instrumentation of the building with specific hardware. The only assumption Walkie-Markie requires is that there exists a WiFi infrastructure in the building that is to be mapped. AP locations do not need to be known; instead APs must merely exist for the system to work.

Walkie-Markie is based on two key observations. First, a modern mobile phone can dead reckon the user's movement trajectory from its inertial measurement units (i.e., IMU sensors, including accelerometer, magnetometer and gyroscope) [21,26,28,36]. The idea is that if sufficiently many users walk inside a building and report their trajectories, we can infer the pathway map. The challenge is that IMU-based tracking is accurate only initially as it suffers from severe drift: rapid error accumulation over time. Moreover, to generate maps at scale via crowdsourcing, we must deal with trajectories from different users, who may start their walks from anywhere, at different stride lengths, varying speed, etc. Second, WiFi networks have been widely deployed, from office buildings to shopping malls. WiFi has been successfully used by fingerprinting-based localization schemes, and combined WiFi and IMU-tracking solutions have also been proposed, e.g., [5,11,26,31,38]. However, there are well-known practical concerns when using WiFi for localization: signals fluctuate significantly during different times of day, different phones can have different receiver gains (i.e., device diversity) [14,34], and readings also vary depending on how people place their phones e.g., in hand, in pocket, or in backpack (i.e., usage diversity) [19].

Walkie-Markie consists of mobile clients on users' mobile phones and a backend service in the cloud. When participants walk, the client collects the rough trajectory information (step count, step frequency, and walking direction) as well as periodic WiFi scan results. The backend service fuses these possibly partial user traces (w.r.t the overall internal pathways) and generates the pathway map.

Central to Walkie-Markie is the *WiFi-defined landmark* (*WiFi-Mark*), which is a novel way to exploit the widely-deployed WiFi infrastructure to establish accurate and stable landmarks, which serve to anchor the various partial trajectories. A WiFi-Mark is defined as a pathway location at which the *trend* of received signal strength (RSS) from a certain AP reverses, i.e., changes from increasing to decreasing, as the user moves along the pathway. We show in this paper that such WiFi-Marks based on the RSS *trend* (instead of *the face RSS value* used in previous works) overcomes the aforementioned challenges in leveraging WiFi signals and yields highly stable and easily identifiable landmarks. WiFi-Marks are determined by the relative physical layout of the AP and the pathway, and are thus location invariant. Moreover, a single AP often leads to multiple uniquely identifiable WiFi-Marks, leading to a higher density of WiFi-Marks.

WiFi-Marks allow us to overcome two key problems in mapping buildings: i) merging the large volumes of crowdsourced (partial) trajectories and ii) bounding the tracking error and drift of IMU sensors. Being location-invariant, WiFi-Marks yield the common reference points for fusing snippets of user trajectories. With more user trajectories, the noise tend to cancel each out, which leads to more accurate displacement measurement between WiFi-Marks. Thus, mapping accuracy gradually improves as more data becomes available. IMU-based tracking suffers notoriously from rapid error accumulation as distance increases. WiFi-Marks also help with the drift problem of IMU-based tracking by bounding the distances between which IMU-based tracking must be relied upon.

Another ingredient of Walkie-Markie is a novel graph embedding algorithm, *Arturia*, that fixes WiFi-Marks to "known" 2D locations respecting the constraints suggested by the user trajectories. The resulting pathway map naturally reflects the physical layout. Arturia differs from existing embedding algorithms in that it uses measured *displacement vectors* as opposed to distances between nodes as input constraints. After WiFi-Marks are properly placed on the 2D plane, the pathway map is generated by connecting the embedded WiFi-Marks with corresponding user trajectories. The obtained pathway maps can be used by users to localize themselves by adding the displacement to the position of the last encountered WiFi-Mark. The pathway maps can also be used to generate other localization maps such as radio maps.

We have implemented Walkie-Markie and evaluated it in an office building and a shopping mall. Our experimental results show that we can achieve mapping accuracy within 3 meters by merging enough user trajectories (each as short as one minute) equaling to 5-6 rounds of walking. The mapping accuracy gradually improves and stabilizes after about 1-2 times more walking time along the same paths. Additional experiments on localization using pathway as well as radio maps produced by Walkie-Markie show that the average and 90 percentile localization errors are 1.65m and 2.9m, respectively, when using displacement from the last WiFi-Mark using the pathway map.

2 Problem and Challenges

Problem Statement: Indoor localization results are meaningful only when associated with corresponding indoor maps (e.g., pathway maps) that geographically reflect the physical layout. However, in this context the availability of such maps has largely been taken for granted, often via assumptions. For instance, IMU-based tracking and localization systems have assumed accurate indoor maps (e.g., floor plans) to constrain drifting; WiFi-based localization systems *further* assume a-priori knowledge about AP positions or a radio signal map [4,13,24]. While there are many existing works trying to

reduce the dependency on such a-priori information (AP locations [6], radio signal map [12, 16, 22, 26, 38]), they all assume a known internal map of the floor, which are often not readily available.

In this paper, we remove this assumption and build an indoor pathway mapping systems without assuming *any* prior knowledge of the building. Our goal is to find a solution that works with existing infrastructure, is applicable to commercial mobile phones, and is "crowdsourceable" so as to scale to a large number of buildings. The only assumption we make is the *mere* existence of a WiFi infrastructure in the buildings to be mapped.

Challenges: Previous work has tried to combine WiFi signals and IMU sensing data [5,10,11,31]. The problem with IMU-based technologies is that they can track a user's trajectory at some accuracy for only a short period of time, and will drift severely as the walking time increases. This makes it hard to align multiple trajectories, and trajectories obtained from different users (with different start points) are even harder to combine into a whole pathway map. Leveraging WiFi also poses well-known challenges. Even though WiFi fingerprints are statistically locality-preserving [16, 24], an AP's coverage area is overly large for the desired accuracy of a useful internal pathway map. Typically, multiple pathways are covered by a single AP. The AP's position is also unknown. Furthermore, other challenges common to WiFi-based localization systems are: i) WiFi signal fluctuations due to ambient interference, multipath effect, and environmental dynamics such as the time-of-day effect; ii) device diversity with different receiver gains at different phones; and iii) device usage diversity caused by people placing their phones differently such as in hand, in pocket, in purse, or in a backpack. We note that usage diversity is rarely mentioned in the literature, but is a real impairing factor.

3 WiFi-defined Landmark

In real life, *landmarks* are often used to give directions. No matter how one detours, once a landmark is encountered, previous errors are reset. Using the same idea, we can leverage landmarks to constrain the drifting in IMU tracking, and to align different user trajectories. However, the challenge is the find landmarks that are perceivable by mobile phones without human intervention. Since mobile phones can sense the WiFi environment in the background, we would ideally like to identify landmarks based on WiFi signal. In this section, we show that–using the concept of *WiFi-Marks*–this is indeed possible in spite of the multitude of challenges mentioned above.

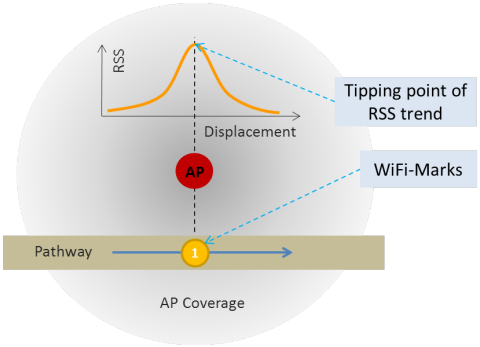


Figure 1: Illustration of WiFi-Marks, as determined by the relative physical layout of the AP and the pathways.

3.1 WiFi-Marks: Concept

Previous work on WiFi-based localization has used the received WiFi signal strength (RSS) directly. It turns out that this is the root cause of the aforementioned problems. The key insight is that significantly more stable landmarks can be obtained from an existing infrastructure by using WiFi signal strength *indirectly*: instead of looking at the face RSS values, we look at the *trend* of RSS changes.

Figure 1 illustrates the basic idea. A user is walking from left to right along a pathway covered by an AP. Initially we see RSS increase as the user moves closer towards the AP. When the user walks past the point from which the distance to AP increases, the RSS trend reverses. In theory, this *RSS trend tipping point* (RTTP) should correspond to a fixed position on the pathway that is closest to the AP in terms of signal propagation.

The key appeal of examining the RSS trend instead of taking individual RSS readings is that it may solve the device and usage diversity problems: no matter what make and model of the phone, what time of the day, and how the phone is kept with respect to its user, the RTTP should occur at around the same location. Through detailed experiments, we argue in Section 3.3 that locations where the RSS trend of a certain AP changes are excellent candidates for landmarks. We call these points WiFi-defined landmark, or short WiFi-Marks (WM) hereafter.

3.2 WiFi-Marks: Identification

As a landmark, each WiFi-Mark should be uniquely identifiable. Depending on how the coverage area of an AP intersects with the pathways, it is possible and in fact quite likely that one AP will generate multiple WiFi-Marks (see Figure 2). Hence while BSSID (the MAC address of the AP) can uniquely identify the master AP, it alone is insufficient to uniquely identify a WiFi-Mark since there can be multiple pathways under the cover-

age of the same AP. Therefore, we need to use additional information to differentiate different WiFi-Marks of the same master AP.

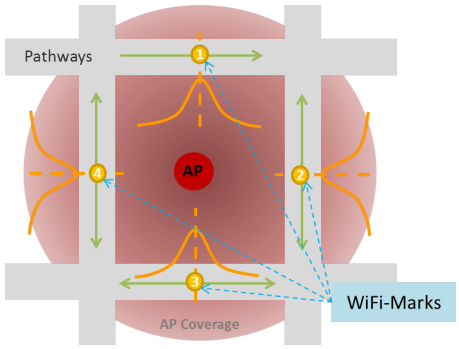


Figure 2: Possibly multiple WiFi-Marks for the same AP.

In Walkie-Markie, we identify a WiFi-Mark by the following three-tuple:

$$WM \triangleq \{BSSID, (D_1, D_2), \mathcal{N}\}$$

where *BSSID* is the ID of the master AP, D_1 and D_2 are the steady walking directions approaching and leaving the RTTP, respectively. They can be obtained from the phone's magnetometer. $\mathcal{N}$ is the set of neighboring AP-s' information, including their BSSID and the respective RSS differences to that of the master AP.

The walking direction information (D_1, D_2) is adopted to differentiate pathways and turns. For example, Figure 3 shows the possible RTTPs for AP_1, under different walking patterns. With directions, we can readily differentiate RTTP 1, {2,3}, 4, and 5. In addition, the direction can be used to disqualify some erroneous RTTP detections when the user makes a U-turn (e.g., RTTP 6 and 7). Not identifying such "U-turn RTTPs", could add significant noise to the system.

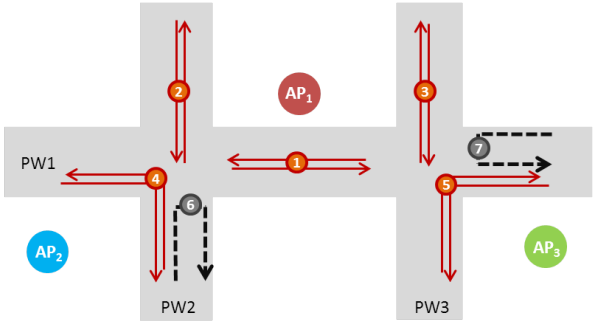


Figure 3: Multiple RTTP possibilities for AP1 under different walking patterns illustrated by arrows.

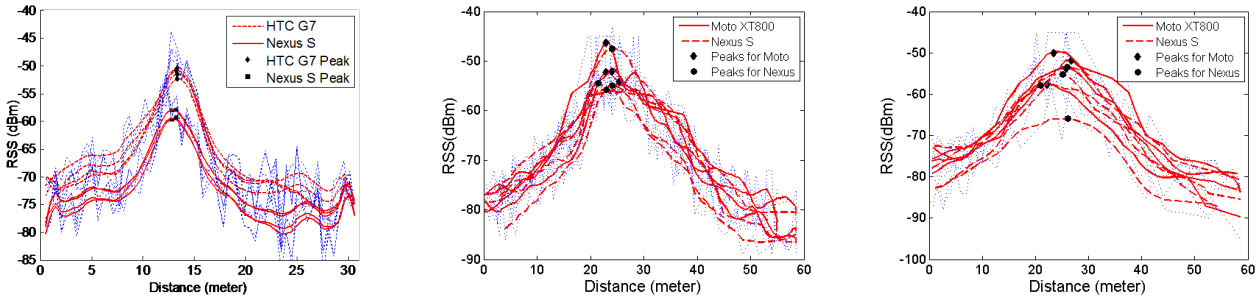
RTTPs with similar (D_1, D_2) can arise from parallel corridors (e.g., RTTP 2 and 3 in Figure 3) or similar turning styles. To further differentiate such RTTPs, we leverage neighborhood AP information. In the same example, RTTP 2 may see AP2 only and RTTP 3 sees AP3 only. Even if they see the exact same set of APs, there is still a good chance that the relative RSS values will be different due to the difference in distance to each AP. Note that it is important to use the RSS *differences* to the master AP's RSS instead of their real RSS values to avoid the device diversity problem. From the radio propagation model [1], it can be verified that RSS differences between multiple APs are not affected by the receiver gain for a device.

Due to sensing noise, D_1, D_2, and $\mathcal{N}$ of a given WiFi-Mark can be slightly different each time the WiFi-Mark is measured. Therefore, we employ a *WiFi-Mark clustering* process (see Section 6). There are further unreliable RTTP detections, such as when a user is not walking straight or steadily (e.g., zigzagging) or when the phone's position changes rapidly (e.g. taken out of the pocket). Our system therefore accepts an RTTP as a WiFi-Mark only if the IMU sensor indicates a stable walking motion and no U-turn is detected during the measurement process.

3.3 WiFi-Marks: Stability

Evaluation Scenarios: The indoor radio environment is complex and often deviates significantly from ideal propagation models. To verify the stability of the RTTPs in practice, we conduct the experiments using different devices (HTC G7, Moto XT800, and Nexus S), at different time of day (morning, afternoon, evening, and midnight), and with the phones held at different body positions (hand, trouser pocket, purse, and backpack). All of these are important factors affecting RSS. In addition, we perform experiments in two buildings to demonstrate the generality of our approach.

We present two sets of experiments. In the first set, we walk and wait, i.e., wait to ensure a complete scan of all WiFi channels before walking to a next collection point. This represents an ideal case. In the second set, we walk continuously at slow or normal speed without waiting for WiFi scans to complete. Figure 4 shows the curves of collected RSS values and the locations of the detected WiFi-Marks. From the figure, we can see that the RSS values from different devices are evidently different, and the same is true for the same device at different time of day, or at different body positions. In contrast, the increasing and decreasing RSS trends are always easily identifiable, and the WiFi-Mark positions are not only highly clustered and stable, but also consistent between the two devices. Taking the normal walking case as an example, the average position deviations are 1.3m and 2.9m for Moto XT800 and Nexus S, re-



(a) Walk and wait. (b) Slow walking. (c) Normal walking.

Figure 4: RSS curves for one AP along a corridor using two phones. Blue dotted lines and red solid lines are the raw and filtered RSS curves (see Section 6). Multiple same type of lines are measurement from different time of day. In (a), all phones were held in hand. In (b) and (c), Moto was held in hand and Nexus S in trouser pocket.

spectively, while the mean center position offset is only 2.7m between devices. In the ideal case, the deviations are even better. The reason is that because of the relatively long WiFi scanning time in today's mobile phone (usually about 1.5s), the user may have already walked a few steps during a scan.

Stability Evaluation: We also conduct controlled experiments in larger areas with more pathways, still using various devices and walking at various speeds and at different times of day. For each different setting we collect data over 5 rounds and calculate the statistical deviation in WiFi-Mark position. We note that the peak RSS value at RTTPs are not all strong, some being as weak as -75 dBm.

Figure 5-(a) shows the cumulative distribution function (CDF) of the deviations for the different settings. We can see that for over 90% of WiFi-Marks, the deviations are within 2.5m, and about 70% are within 1.5m in all cases. We further study whether WiFi-Marks detected with different settings are consistent, using the center offset of WiFi-Mark clusters. Figure 5-(b) shows the CDF of the center offsets. They are indeed consistent: over 95% of the offsets are within 2.5m and over 75% are within 1.5m. These results demonstrate that WiFi-Marks are stable and robust across various dimensions, and thus have ideal properties to be landmarks for our indoor pathway mapping purpose.

4 Walkie-Markie: Overview

With WiFi-Marks, we now have the common reference points for fusing crowdsourced user trajectories together. Walkie-Markie consists of a client–an application running on users' mobile phones–and the backend service running in the cloud. The overall architecture is shown in Figure 6.

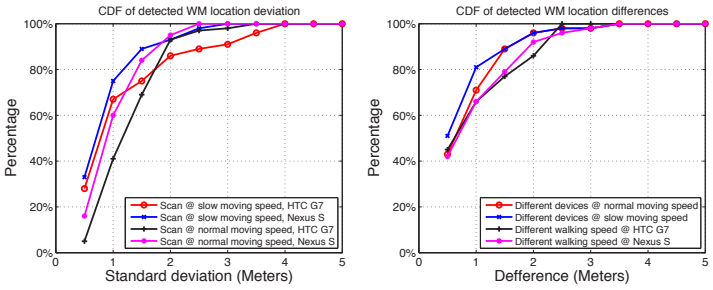
A Walkie-Markie client works as follows: a background motion state detection engine monitors users' motion states periodically. When the user is detected in *walking* state, IMU-based tracking is activated and the instantaneous walking frequency and direction of each step is recorded for displacement estimation. At the same time, WiFi signal scanning is performed opportunistically. If a WiFi signal is detected and the device has not associated with an AP, the WiFi-Mark detection process is activated. Information about the detected WiFi-Marks and estimated displacements between neighboring WiFi-Marks are stored, and later sent to the backend service.

The Walkie-Markie backend service listens to WiFi-Mark updates from all clients. Upon receiving WiFi-Mark updates, it examines if their master APs are new or already existing. Updates with new master APs are recorded and aged to mitigate the impact of transient APs (e.g., mobile APs). For existing ones that are old enough, their neighborhood consistency is further checked to ensure they are not relocated APs, which would be treated as new APs. Then a clustering process is executed to cluster different detections of the same actual WiFi-Marks. Each cluster is then assigned one coordinate by the Arturia engine. Finally, with WiFi-Marks positioned at the right places and user trajectories connecting them, the backend service can generate the desired pathway maps.

5 WiFi-Mark Positioning

WiFi-Marks (or landmarks in general) serve their purpose as a reference points only once we can place them at a known location. For this reason, we need to assign coordinates to WiFi-Marks, which is a classical node embedding problem in the network coordinate and localization literature.

Distance vs Displacement: Previous node embedding work has unanimously assumed scalar distances (e.g., via a direct distance measurement or the shortest path) between nodes [9, 23, 30]. However, in our case, users may



(a) CDF of WM position deviation (b) Consistency of WM positions

Figure 5: Statistics on WM positions.

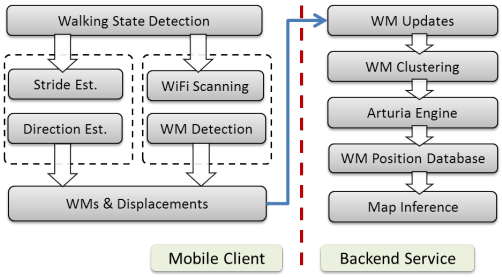


Figure 6: Walkie-Markie system architecture.

not always take the shortest path and, in fact, the internal floor layout may even prevent people from taking shortest paths (e.g., two nearby WiFi-Marks blocked by a wall or a locked door). If multiple paths exist, taking different paths will lead to different distances. These factors often lead to severe violations of the triangle inequality, which lies at the heart of existing embedding algorithms. Consequently, using distances between WiFi-Marks is insufficient and the *displacement vector* (i.e., both the distance and the direction, obtainable from IMU sensors) between WiFi-Marks has to be used.

Using direction information in addition to distance is fundamental, because it can largely avoid the "fold-freedom" problem of the embedding process [25], and dismiss flip and rotational ambiguities. The *only* remaining translational ambiguity can be fixed by fixing any anchor point with an absolute location (e.g., entrances or window positions of a building with GPS readings). In addition, using direction information also requires fewer measurements: only N unique displacement measurements are required to localize N WiFi-Marks, whereas $3N-6$ unique measurements would be required when using distances only (in which case the results would still suffer from flip and rotational ambiguities). Thus, using displacement vectors enables faster bootstrapping and is highly desired for a crowdsourcing system.

5.1 Arturia Positioning Algorithm

In our system, a major challenge is the inaccuracy of IMU-based displacement measurements (e.g., errors in stride length and/or direction estimation). To compensate these errors, we design a new embedding algorithm, *Arturia*, that handles noisy IMU measurements and assigns optimal coordinates to WiFi-Marks. Arturia is based on the *spring relaxation* concept, where each edge of the graph is assumed as a spring and the whole graph forms a spring network.

Building the Graph: An edge (hence, a spring) is added between two specific WiFi-Mark nodes as long as there exists a real user trajectory in between. The rest length of the spring (i.e., the constraint) is the real displacement measurement from user trajectory. Multiple edges between a pair of nodes are possible if there exist multiple user trajectories. In this way, we ensure that more frequently encountered WiFi-Marks will have more accurate coordinates as compared with the alternative strategy that uses a single average edge.

Realizing the Graph: With the spring network, our goal is to minimize the overall residual potential energy E, which is a function of the discrepancy between the calculated distance (i.e., actual length of the spring) and the real measurement (i.e., rest length of the spring). Our solution is to adjust the node's position as if it were pushed or pulled by a net force from all connecting neighboring springs. Arturia works as follows:

Initialization: We may randomly assign all nodes's initial coordinates, or simply to the origin. But for updates due to new incoming data, the previous coordinates are used for faster convergence and better consistency, i.e., minimal adjustment to the previous graph.

Iteration: At each iteration, adjust the coordinates for each node according to the compound constraints of the neighboring nodes. Let $\hat{p}_i$ be the current coordinate of node i. We have $\vec{d}_{i,j} = \hat{p}_i - \hat{p}_j$ as the current displacement vector between node i and a neighboring node j. Assume there are $N_{e,i,j}$ real measurement constraints between node i and j, and let $\vec{r}_{i,j,k}$ be the k^{th} constraint. Then the adjustment vector is calculated as

$$\vec{\varepsilon}_{i,j} = \sum_{k=1}^{N_{e,i,j}} (\vec{r}_{i,j,k} - \vec{d}_{i,j}) \qquad (1)$$

The gross adjustment vector $\vec{F}_i$ is obtained by summing up $\vec{\varepsilon}_{i,j}$ over all neighboring nodes, i.e., $\vec{F}_i = \sum_j \vec{\varepsilon}_{i,j}$. Then, node i's coordinate is updated as $\hat{p}_i = \hat{p}_i + \vec{F}_i$.

The step size of the adjustment (i.e., $|\vec{F}_i|$) plays a critical role in the convergence speed: large adjustment steps may lead to oscillation while small adjustments will converge slowly, as also observed in [9]. To obtain a suitable step size, we empirically amortize the adjustment vector

according to $N_{e,i}$, the total number of edges to all neighbors of node i. That is, $\vec{F}_i = \frac{1}{N_{e,i}} \sum_j \vec{\varepsilon}_{i,j}$.

Termination: For each node i, the local residual potential energy E_i is calculated as $E_i = \sum_j |\vec{\varepsilon}_{i,j}|^2$. System residual potential energy is then $E = \sum_i E_i$. This value tends to increase with additional edges of the spring network. To obtain a universally applicable termination criterion, we use the normalized potential energy $\bar{E} = E/N_e$ with N_e being the total number of constraining edges. The iteration will terminate when the change of $\bar{E}$ falls below a small pre-determined threshold.

Algorithm Comparison: The spring relaxation concept has previously been adopted, e.g. in [9, 15, 25]. The major difference is that the local adjustment (i.e., $\vec{F}_i$) in each iteration has *direction* information and will always move closer to the target coordinates in Arturia. This is not the case in other algorithms where the moving direction is calculated based on the noisy, intermediate coordinates. Figure 7 illustrates this difference between Arturia and the Vivaldi [9] algorithm for an intermediate adjustment step to Node 3. We can see that in Arturia, the net force of the adjustment points directly to Node 3's target position, while in Vivaldi it does not. The reason is that the constraints in Arturia are *displacement vectors* (e.g., $\vec{r}_{3,1}$ and $\vec{r}_{3,2}$) with direction information, while in Vivaldi they are scalar distances (e.g., $|\vec{r}_{3,1}|$ and $|\vec{r}_{3,2}|$).

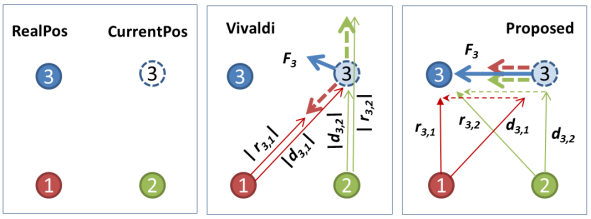


Figure 7: Illustration of an intermediate adjustment step of Vivaldi [9] and Arturia.

5.2 Arturia Evaluation

We evaluate Arturia with simulations. We randomly deploy N nodes in a 100×100 square area. For each node, we build n edges to n random neighboring nodes. For each edge, the direction is adjusted by a random number within ± 30 degrees, while the distance (i.e., the magnitude of displacement) is randomly adjusted by within ± 10 percent. These numbers reflect the real displacement estimation error ranges.

Anti-folding Capability: As mentioned, using direction helps to avoid "fold-freedom" issues. We demonstrate this by comparing the snapshots of intermediate steps of Arturia against those of Vivaldi and AFL (see Figure 8). We see that after 100 iterations, the nodes are still heavily folded in Vivaldi. AFL is better than Vivaldi in shape, but at a wrong scale. For Arturia, the nodes are almost in correct positions after only 30 iterations.

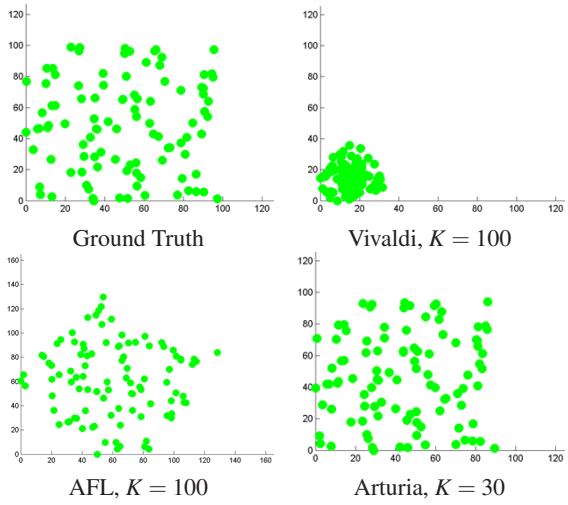


Figure 8: Snapshots of node positions at the different iterations for Vivaldi, AFL and Arturia.

Speed and Accuracy: We study the convergence speed and the resulting accuracy of different algorithms by varying the parameters N and n. Each experiment is repeated 10 times and average results are reported. Note that in the simulation, we have used the magnitude of displacement as the distance for Vivaldi and AFL to ensure the obeyance of triangular inequality, i.e., all nodes are mostly localizable.

The speed is measured as the number of iterations. For the accuracy metric, we adopt the Global Energy Ratio (GER) because it captures the global structural property [25]. GER is defined as the root-mean-square normalized error value of the node-to-node distances, i.e., $GER = \sqrt{\Sigma_{i,j:i<j} \hat{\varepsilon}_{ij}^2 / (N(N-1)/2)}$ where N is the total node number and $\hat{\varepsilon}_{ij} = |\Delta\vec{d}_{ij}|/|\vec{d}_{ij}|$ is the normalized node distance error.

Table 1 shows the results. We see that the proposed Arturia algorithm is significantly better than the other two algorithms in terms of both convergence speed and accuracy. In general, with higher connectivity, both speed and accuracy improve for all three algorithms. This is due to larger damping effects resulting from more densely interconnected springs. However, even with dense connectivity, the accuracy of Vivaldi is poor because of heavy folding. AFL works better by finding better initial positions. In our target scenario, the node connectivity cannot go very high since there will rarely be direct displacement measurements between faraway WiFi-Marks. This highlights the advantage of Arturia in

N	n	Speed (Iterations)			Accuracy (GER)		
		Viv.	AFL	Art.	Viv.	AFL	Art.
100	4	319k	193k	763	.687	.241	.0091
	6	38k	27k	450	.660	.106	.0072
	8	11k	2244	232	.615	.015	.0061
	10	6954	971	170	.614	.012	.0056
200	4	340k	334k	1552	.745	.279	.0068
	6	42k	19k	706	.736	.060	.0053
	8	20k	3299	441	.710	.012	.0049
	10	10k	1552	339	.699	.010	.0046

Table 1: Speed and Accuracy comparison of Vivaldi, AFL, and Arturia. N is the node number and n is the node connectivity degree.

the context of Walkie-Markie.

6 System Implementation

We have implemented the Walkie-Markie system, with mobile client on Android phones and backend services as Web Services. In this section, we detail a few key components.

WiFi-Mark Detection: In mobile client, the collected RSS value is first smoothed over a 9-point weight window in a running fashion to detect WiFi-Marks. The weight window is empirically set as a triangle function ($\{0.2, 0.4, 0.6, 0.8, 1, 0.8, 0.6, 0.4, 0.2\}$). We tested other window functions (e.g., cosine, raised cosine) and found not much difference in detection accuracy. Then, the trend detection is done by taking derivatives of the smoothed RSS curves, i.e., the differences between neighboring points. The consecutive positive and negative spans of the differences are identified and the corresponding walking directions are checked. If there are no U-turns and the trend change is significant as controlled via a threshold (e.g., 5 dBm), the point with the peak (filtered) RSS during the trend transition is selected as a WiFi-Mark.

Displacement Estimation: Displacement between WiFi-Marks is estimated from user trajectories by accumulating the displacement of each step. Step displacement carries stride length and walking direction and is captured by IMU sensors. Many techniques exist for stride length estimation [17, 29, 32]. We chose a simple frequency-based model by Cho *et al* [7]: $stride\_len = a \cdot f + b$ with f being the instantaneous step frequency, and a, b being parameters that can be trained offline. However, model parameters are specific to a user's walking conditions, e.g., parameters trained from wearing sport shoes will not work well when wearing high heels. Improper parameters will lead to large estimation error.

Interestingly, leveraging common WiFi-Marks among user trajectories, we can avoid the error-prone stride length estimation and instead rely on simpler and more robust *step counting* under regular walking, which can be easily be done from the regularity of the IMU data. We first randomly select a user and treat her stride length as the benchmark unit (BU). We then normalize other users' stride against hers using partial trajectories between common WiFi-Marks and obtain a normalization factor θ. This normalization process is transitive. Ultimately all users will normalize their traces to the same BU and obtain their respective θs. Then we obtain the average normalization factor $\bar{\theta}$. The product of $\bar{\theta}$ and the BU will be the real stride length of the "average" user, to which we can assign the demographic average stride length.

Walking Direction Estimation: We use the magnetometer and the gyroscope to obtain the walking direction and the turning angles, similar to [18, 21]. Unlike step detection and stride length that is determined on a per-step basis, the direction of each step needs to be determined by considering those of neighboring steps because magnetometer readings are sometimes not stable due to disturbance of local building construction or appliances, and the gyroscope may drift over time. In our implementation, we simply discard portions of magnetometer data where drastic changes occur, and rely on the gyroscope to decide whether there is a direction change in that period. For the portions with stable magnetometer readings, we use a Kalman Filter to combine the magnetometer and the gyroscope readings to tell the user's walking directions.

WiFi-Mark Clustering: The backend service receives many crowdsourced trajectories and WiFi-Mark reports. Due to sensing noise and user motion, the same actual WiFi-Mark may be reported slightly differently in directions (D_1, D_2) and neighborhood ($\mathcal{N}$). We design a clustering process to detect the same actual WiFi-Mark: we first classify reported WMs with the same BSSID using D_1 and D_2. To accommodate sensing noise, the directions are considered the same if they are within ± 20 degrees. For those WMs with same BSSID and similar directions, a bottom-up hierarchical clustering process as in [6, 24] is applied on the neighborhood set $\mathcal{N}$. Initially, each WM is one cluster. Then the closest clusters are iteratively merged if their inter-cluster distance is smaller than a pre-determined threshold, which is set to 15 dBm as recommended in [24].

The inter-cluster distance is the average distance between all inter-cluster pairs of WMs. The distance between two WMs is defined over the RSS of the sensed APs ($\mathcal{A}$) as follows:

$$D_N(\vec{\mathcal{A}_i}, \vec{\mathcal{A}_j}) = \sqrt{\sum_{n=1}^{K} (a_n^i - a_n^j)^2 / K}$$

where $\vec{\mathcal{A}}_i$ are the RSS *differences* (to ensure device indifference) between the master AP and neighboring APs at WM_i, and K is the total number of unique APs detected at the two WMs. For orphan APs appearing in one WM's neighborhood but not the other, the RSS difference is set to peak RSS of the master AP minus -100 dBm. Finally, each WiFi-Mark cluster is treated as one node in Arturia and assigned one coordinate. All WiFi-Marks in the same cluster have the same coordinate.

Pathway Map Generation: With WiFi-Marks and connecting user trajectories, we design the following expansion and shrinking procedure to generate the pathway map systematically. Initially, user trajectories are partitioned into snippets delimited by WiFi-Marks. Snippets with U-turns are filtered out. The remaining trajectory snippets are calibrated by proportionally adjusting the length and direction of each step using the WiFi-Marks' coordinates assigned by Arturia (affine transformation). For each calibrated trajectory snippet, we first draw it on a canvas and further expand it to a certain width (i.e., from line to shape). Pixels being occupied are weighted differently according to their distances to the center line: the closer the pixels, the higher the weight. Due to the multiplicity of user trajectories, there may exist multiple snippets connecting the same two WiFi-Marks. Thus, expanded snippets will overlay together and the weight of overlapping pixels are summed up. The expansion process will result in a fat pathway map. A shrinking process is then applied to prune away those outer pixels whose weights are less than a threshold. As some WiFi-Marks may be encountered more frequently than others, we adapt the threshold as a percentage to the maximum weight in the local neighborhood. Finally, we remove isolated pixels and also smooth the edges of the resulting shrunk pathway map.

Note that the pathway map generated from above expansion-shrinking process is a bitmap. It is also possible to generate a vector pathway map as all the turns can be effectively determined from user trajectories. We adopt bitmap in this paper for its immediacy in visually reflecting the quality of users' trajectories.

Practical Considerations: In our implementation, we have considered other important issues to build a practical crowdsourcing system.

Robustness: We have designed two mechanisms to improve the robustness of the system. First, the backend service implements an enrollment selection mechanism. WiFi-Marks from new master APs are recorded and will be incorporated into the Arturia positioning engine only when the AP becomes sufficiently aged. This is to counter transient WiFi-Marks, e.g., those caused by mobile APs or WiFi hotspot created on mobile phone through tethering. Relocated APs are detected via neighborhood (carried in WiFi-Mark reports) consistency and treated as new APs. Second, to mitigate the impact of outlying WiFi-Marks, e.g., resulting from transient mobile AP or wrongly detected due to magnetometer malfunction, we enroll a WiFi-Mark cluster only when it has a sufficient number of members (e.g., three).

Energy consumption: IMU-sensing consumes little energy, especially at low sampling rate (e.g., 10Hz in our case). Our preliminary test shows that 10Hz IMU sensing shortens the depletion time of a fully charged battery from 18.3 hours to 17.8 hours. We reduce data communication to the server by performing step detection and WiFi-Mark detection entirely on the mobile phone. The final communication data rate is about 1KB every 100 steps. Note that it can be delayed and piggybacked on other network sessions. The major energy consumption is from WiFi scanning. To work around, our client triggers WiFi scanning only when the user is walking (detected from low duty cycled IMU sensors), and we task a user to collect just a few minutes of walking data. As shown in Section 7, even short trajectories can still be used to infer pathway maps.

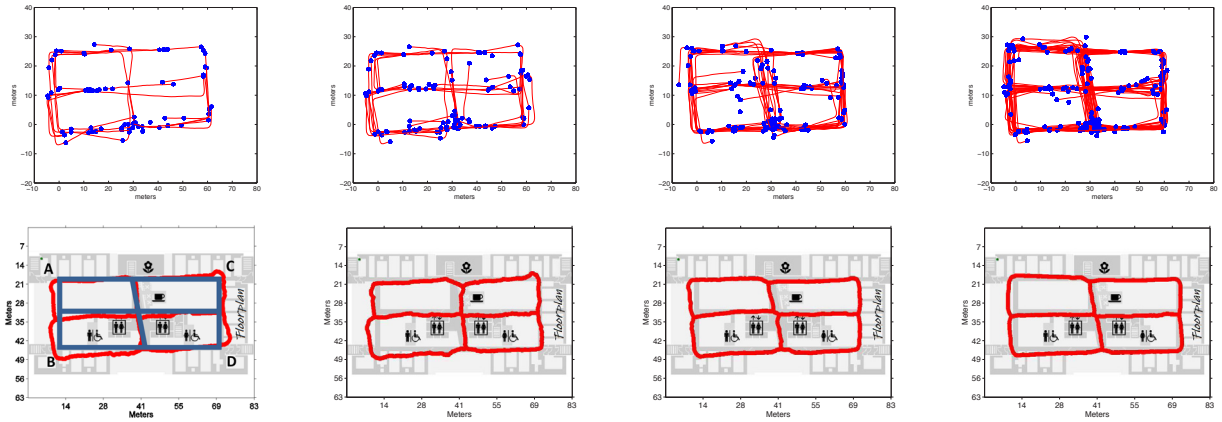
7 Walkie-Markie System Evaluation

7.1 Visual Comparison

Before presenting quantitative evaluation results, we first visually examine the inferred pathway map with the ground truth or reference floor plan. This will give us a general feel for Walkie-Markie's practicality.

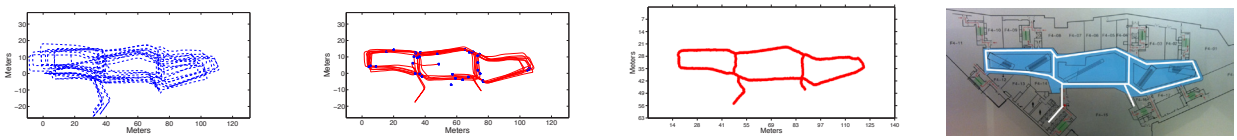
An Office Floor: We first show the study in our office floor for which we have the groundtruth floor plan. The internal layout consists of meeting rooms, offices, cubicle areas, and relatively large open areas in the middle. The experiment floor size is 3,600m^2 and the total internal pathway length is 260m. Figure 9 shows the aligned user trajectories and the inferred pathway maps under different amounts of user trajectory data. The stars in user trajectories are the detected WiFi-Marks. As expected, the quality of the resulting pathway map improves with more user data. After 50 minutes of random walk, the resulting map is already very close to the real map shown in the bottom-left figure.

A Shopping Mall Floor: We also study a nearby shopping mall. There is no managed WiFi LANs, but many isolated WiFi islands deployed by coffee shops or from POS machines. The floor has an irregular layout and the internal pathway length is roughly 310m. We walked about 10 rounds for about 40 minutes with a Nexus S phone. The results are shown in Figure 10. The first two figures show the raw IMU-tracked user trajectories



(a) after 20min walk. (b) after 30min walk. (c) after 50min walk. (d) after 100min walk.

Figure 9: Aligned user trajectories and generated pathway maps at different amount of user trajectories.



(a) normal IMU-tracking (b) with WM alignment (c) Inferred pathway map (d) Picture from flyer.

Figure 10: The picture and generated pathway map for a real shopping mall.

and those aligned with WiFi-Marks. The third figure shows the inferred pathway map. Unable to obtain a groundtruth floor plan, we took a picture of an emergency guidance map and highlighted the pathways in the last figure. We see that the pathway map generated by Walkie-Markie is visually very close to the real one.

7.2 Quantitative Evaluation

We conduct experiments in our office building, for which we have the groundtruth floor plan.

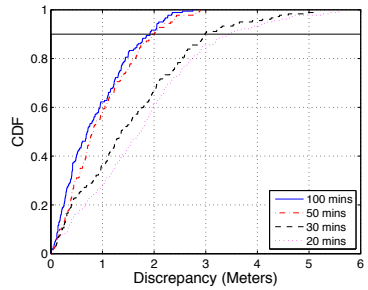
Data Collection: We have collected data from seven users, six male and one female, with heights range from 158cm to 182cm. A stride length model is trained for each user. We asked them to walk normally and cover all the path segments in each round, but they could start anywhere. Three phone models (Nexus S, HTC G7, and Moto XT800) were used. The phones were held in hand in front of body, hip-pocket, and also a backpack. In total, the users walked 30 rounds for about two hours.

In real crowdsourcing scenarios, users may walk only a portion of all pathways, or we may need to discard portions with irregular walking, or a user may only want to be tasked for a short time for consumption of energy consumption. To simulate these constraints and see if short trajectories are still useful, we chop the complete user trajectories into one-minute snippets, and randomly select a certain number of such snippets to infer the map. Results reported below are averaged over 10 such experiments.

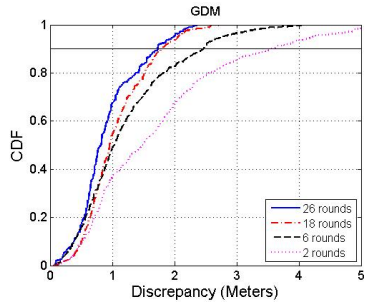
Performance Metrics: To quantify the quality of the inferred pathway map, we use the following metrics.

- *Graph Discrepancy Metric* (GDM): This metric reflects the differences in the relative positioning among anchor nodes, i.e. singular locations such as crosses or sharp turns. Like GER, we compare the Euclidean distances among all node pairs using coordinates from respective maps.
- *Shape Discrepancy Metric* (SDM): This metric quantifies differences between the shape of inferred paths and real ones. Path segments between corresponding anchor nodes are uniformly sampled to obtain a series of sample points. The metric is defined as the distance between corresponding sampling points. Note the inferred map needs to be registered to the real map first by aligning at some anchor nodes.

Mapping Accuracy: Figures 11-(a) and (b) show the cumulative distribution (CDF) of GDM from different amount of trajectory data. We can see that the geometric layout of all anchors are well preserved with only 2-hour walking data. The maximum difference in distances between corresponding node pairs is about 3 meters, and the 90 percentile difference is around 2 meters. We also observe that the performance improves as more data becomes available. In addition, an accurate pathway

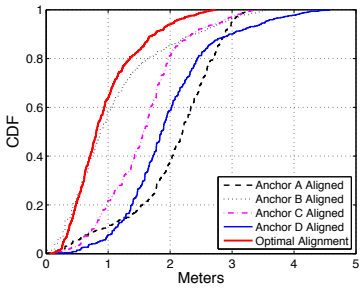


(a) Random snippets

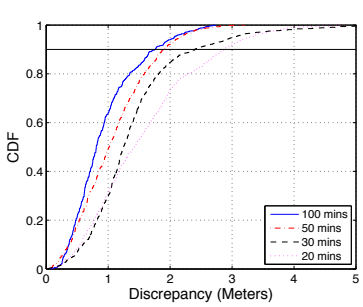


(b) Complete rounds

Figure 11: CDF of GDM.

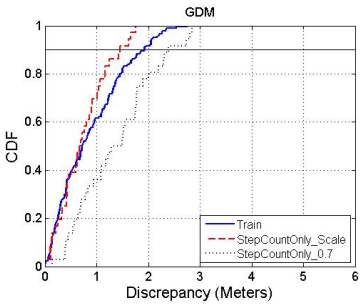


(a) Intuitive alignments

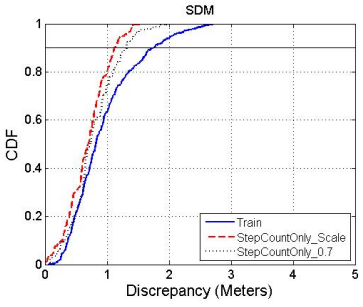


(b) Optimal alignment

Figure 12: CDF of SDM.



(a) GDM



(b) SDM

Figure 13: System performance using step count only.

map can be built from trajectories as short as one-minute walking, as long as we can obtain sufficiently many of them.

Comparing the curves with similar walking time (e.g., 100min vs 26 rounds) in the two subfigures, we can see that using complete trajectories leads to better performance. This is because chopping the walks into snippets reduces the displacement measurements between WiFi-Marks. In general, longer trajectories yields better performance.

In measuring SDM, we have different options to align the inferred map to the real map to fix the only remaining translational ambiguity. In reality, such alignment can be automatically performed by leveraging user trajectories that enter or leave the building. Here, we study the results by aligning at any outermost anchor point (e.g., Points A, B, C, D in the bottom-left figure in Figure 9), and also an optimal alignment at the geometric center of all anchors. In all experiments below, we have obtained 10 sample points on each path segment between two neighboring anchors.

Figure 12-(a) shows the CDF of SDM using 100 one-minute snippets. We can see that aligning at different points indeed leads to different performance. Nevertheless, the maximum difference among all the five alignment trials is small, within 1.3 meters. In the remainder of the evaluation, we use the optimal alignment. From Figure 12-(b), we see that the shape of inferred pathways agrees well with the shape of real ones. When over 50 minutes of walking data is used, the maximum path discrepancy is within 2.8 meters, and the 90 percentile error is within 1.8 meters.

Step Count Only: We stated above that Walkie-Markie can avoid error-prone stride length estimation. To verify this claim, we use only the direction and step count from the same set of user trajectories. Figure 13 shows the results. Since we do not know the demographic average step length, we scale the resulting shape to best fit the ground truth. This gives the upper bound of system performance. We also simply assign 0.7m as the demographic average step length and obtain the results. From the figure we can see that even using step count only leads to high accuracy maps. Comparing with the curve using the trained stride length model, we can see that the 90 percentile GDM is only slightly worse (within 0.4m) and the 90 percentile SDM is actually better by about 0.4m.

Impact of AP Density: Our office floor has a relatively dense AP deployment, about 21 APs covering an area of 3,600m^2. It is natural to conjecture that the performance of Walkie-Markie may be highly affected by the AP density. To study this impact, we emulate sparse deployments by randomly blanking out a certain percentage of APs, i.e., eliminating all the WiFi-Marks defined by those APs and their appearances in other WiFi-Mark's neighbor AP list.

Figure 14 shows the results with varying percentage

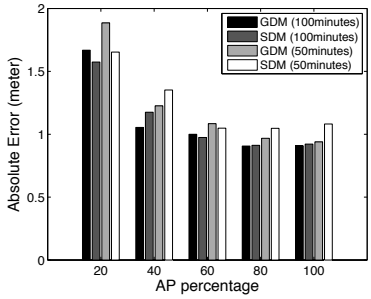


Figure 14: Impact of WiFi AP density.

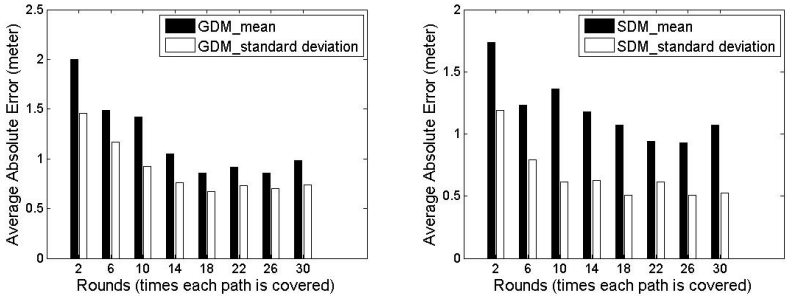


Figure 15: GDM and SDM statistics under different amount of trace data.

of remaining APs. In general, the performance degrades when the number of AP decreases. But for a dense deployment like our office building, the number of APs is more than enough for a good result. The result does not suffer if AP density is reduced to 40%. And even a further reduction to 20% degrade the mapping accuracy only slightly.

System Agility: We are also interested in learning how *agile* Walkie-Markie can construct a useful internal pathway map. System agility reflects the adaptation capability to the internal layout changes of a building. It is measured by the achievable GDM and SDM under different amount of user trajectories incorporated into the system. Figure 15 shows that both discrepancy metrics decrease with more data input, and the system converges quickly: with about 5 to 6 rounds of trajectories (i.e., visits per path segment), a highly accurate pathway map can already be inferred.

8 Application to Localization

Radio Map as Side Product: In Walkie-Markie, WiFi fingerprints are collected when the users walk. When the internal pathway map is generated, the position of each user step can be obtained from the calibrated walking trajectory. With reference to the timestamps of WiFi scans and steps, we can easily interpolate the position of each WiFi scan. As a result, we can generate a dense WiFi fingerprint map for free.

Localization: Both the resulting internal pathway map and the radio map can be used for localization purpose. For the former, we can localize a user by tracking the relative displacement since the last WiFi-Mark encountered, whose position is known. For the latter, we can apply any WiFi fingerprinting-based method such as the RADAR localization system [4]. For evaluation, we walked one round along the pathway in the office floor. During walking, we ensured every step to be at boundaries of carpet tiles. Thus fingerprints are collected at half-meter (i.e., the tile size) interval and their groundtruth positions are also known. We compare the localization results in Figure 16. We can see that Walkie-Markie outperforms RADAR, and more interestingly, the localization error is bounded. Quantitatively, the average and 90 percentile localization errors are 1.65m and 2.9m for Walkie-Markie, and 2.3m and 5.2m for RADAR. We note that the resulting accuracy is comparable with that reported in Zee [26], and slightly better than that from LiFS [38].

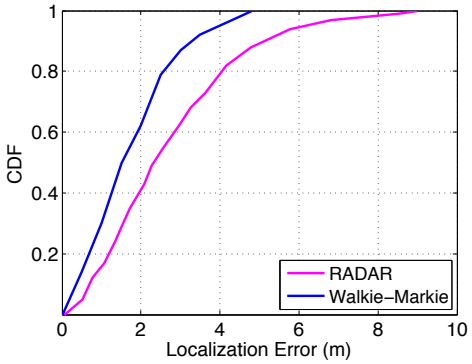


Figure 16: Localization results of Walkie-Markie and RADAR in an office floor, using crowd sourced map.

9 Discussion

Open Area: Our system works well for normal indoor pathways that are typically narrow (say a few meters), which helps ensuring regular user motion. For large open areas, the performance depends on how users walk. If most users walk along roughly the same path (e.g., from one entrance to another), Walkie-Markie will still work. In general, however, the performance may deteriorate as users may walk arbitrarily, which will cause noisy WiFi-Mark detection and clustering. For wide pathways, the inferred map tends to be thinner than the real ones. This is because we have assumed a point representation of a WiFi-Mark cluster, and we have also assumed the pathway to be around 2-meter wide pruning outer pixels in the shrinking process. We note that WiFi-Mark clusters

from wider pathway segments tend to be more diverged than those from thinner ones, we may leverage this fact to estimate the pathway width.

Multiple Floors: Users may walk across different floors using either elevators, escalators, and stairs. These motion states can be discriminated using accelerometer with advanced detection mechanisms [20,35], and can thus be excluded in the WiFi-Mark detection. Interestingly, these functional areas may serve as landmarks as they are stable and reliably detectable via phone sensors. Thus, they can also be incorporated into the Walkie-Markie system, and treated in the same way as WiFi-Marks by the Arturia engine. To discriminate different functional areas of the same type, we can use the covering WiFi APs.

Dedicated Walking *vs* Crowdsourcing: While Walkie-Markie is crowdsourcing-capable, it can also be used by dedicated or paid war-walkers. Dedicated walkers can walk longer and better traces, which leads to a higher efficiency in generating the desired maps (as shown in Figure 11).

10 Related Work

Although we focus on internal pathway mapping, Walkie-Markie is essentially a system of simultaneous localization and mapping (SLAM), which is heavily studied in the robotics field [33]. SLAM methods typically rely on visual landmarks or obstacles detected by camera, sonar or laser range-finders and on accurate kinematics of robots [2]. FootSLAM [28] uses shoe-mounted inertial sensors to construct the internal map. PlaceSLAM [27] further incorporates *manually* annotated places. In contrast, Walkie-Markie requires no special hardware and uses IMU sensors on commercial mobile phones, and requires no human intervention, which is necessary for a crowdsourcing system.

Escort [8] navigates users via the map built from other users' trajectories and instruments audio beacons to constrain IMU-tracking drift. Unloc [35] further explores various types of natural landmarks detectable from sensor readings, including the landmarks from WiFi networks. Their WiFi landmarks are determined as locations least similar (with ratio of common APs as the similarity metric) to all other places. Walkie-Markie does not need to instrument the environment, and uses the RSS trend to detect WiFi-Marks. This idea makes it robust to signal fluctuations, device diversity, and usage diversity, whereas how Unloc handles such practical issues was not reported. The detection is much simpler. In addition, unlike Unloc where multiple APs may determine one WiFi landmark, one AP may determine multiple WiFi-Marks in Walkie-Markie. Thus, we are able to find significantly more WiFi-Marks (e.g., over 100 WMs in one floor) than Unloc (e.g., around 10 WiFi landmarks and overall 40 landmarks in one building). One recent work [19] also exploits the point of maximum RSS, which bears similarity to WiFi-Mark. However, instead of exploiting it as a landmark, they use it to switch between two location inference modules. A dedicated training stage is required to obtain the locations of such maximum RSS points. Walkie-Markie builds the pathway map without pre-training.

There are several papers that combine WiFi and IMU-tracking for mapping purpose. WiSLAM [5] seeks to construct the WiFi radio map and uses the RSS values to differentiate different paths. WiFi-SLAM [11] uses a Gaussian process latent variable model to build WiFi signal strength maps and can achieve topographically-correct connectivity graphs. SmartSlam [31] employs inertial tracing, a WiFi observation model and Bayesian estimation method to construct the floor plan. LiFS [38] and Zee [26] seek to reduce efforts in generating the radio map, with the necessary aid of the actual floor plan. All these work has exploited the WiFi signal in the same way as other WiFi-based localization methods, and thus still face the same challenges, namely WiFi signal fluctuations, device diversity and usage diversity. Again, Walkie-Markie avoid such challenges by using RSS trend instead of face values.

11 Conclusion

We have presented the design and implementation of Walkie-Markie – a crowdsourcing-capable pathway mapping system that leverages ordinary pedestrians with their sensor-equipped mobile phones and builds indoor pathway maps without any a-priori knowledge of the building. We propose WiFi-Marks–defined using the tipping-point of an RSS trend–to overcome the challenges common to WiFi-based localization. Its location-invariant property helps to fuse user trajectories and make the system crowdsourcing-capable. We also present an efficient graph embedding algorithm that assigns optimal coordinates to the landmarks through a spring relaxation process based on displacement vectors. With the located WiFi-Marks and user trajectories, highly accurate pathway maps can be generated systematically. Our experiments demonstrate the effectiveness of Walkie-Markie.

12 Acknowledgement

We thank all the reviewers for their insightful comments and valuable suggestions, and Dr. Suman Banerjee for shepherding the final revision of the paper.

References

[1] Log-distance path loss model. http://en.wikipedia.org/wiki/Log-distance_path_loss_model.

[2] D. Amarasinghe, G. Mann, and R. G. Gosine. Landmark detection and localization for mobile robot applications: A multisensor approach. *Robotica*, 28:663–673.

[3] M. Azizyan, I. Constandache, and R. Roy Choudhury. Surroundsense: mobile phone localization via ambience fingerprinting. In *MobiCom '09*.

[4] P. Bahl and V. N. Padmanabhan. Radar: An in-building rf-based user location and tracking system. In *Infocom '00*.

[5] L. Bruno and P. Robertson. Wislam: Improving footslam with wifi. In *IPIN '11*.

[6] K. Chintalapudi, A. Padmanabha Iyer, and V. N. Padmanabhan. Indoor localization without the pain. In *MobiCom '10*.

[7] D.-K. Cho, M. Mun, U. Lee, W. J. Kaiser, and M. Gerla. Autogait: A mobile platform that accurately estimates the distance walked. In *PerCom 2010*, pages 116–124.

[8] I. Constandache, X. Bao, M. Azizyan, and R. R. Choudhury. Did you see bob?: human localization using mobile phones. In *MobiCom '10*.

[9] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Sigcomm '04*.

[10] F. Evennou and F. Marx. Advanced integration of wifi and inertial navigation systems for indoor mobile positioning. *EURASIP J. Appl. Signal Process.*, pages 1–11, January 2006.

[11] B. Ferris, D. Fox, and N. Lawrence. Wifi-slam using gaussian process latent variable models. In *IJCAI'07*.

[12] Y. Gwon and R. Jain. Error characteristics and calibration-free techniques for wireless lan location estimation. In *MobiWac '04*.

[13] A. Haeberlen, E. Flannery, A. M. Ladd, A. Rudys, D. S. Wallach, and L. E. Kavraki. Practical robust localization over large-scale 802.11 wireless networks. In *MobiCom '04*.

[14] A. Hossain, Y. Jin, W. Soh, and H. Van. Ssd: A robust rf location fingerprint addressing mobile devices' heterogeneity. *Mobile Computing, IEEE Transactions on*, PP(99):1, 2011.

[15] A. Howard, M. Mataric, and G. Sukhatme. Relaxation on a mesh: a formalism for generalized localization. In *Proc. IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems.*, volume 2, pages 1055 –1060, 2001.

[16] Y. Ji, S. Biaz, S. Pandey, and P. Agrawal. Ariadne: a dynamic indoor signal map construction and localization system. In *MobiSys '06*.

[17] A. R. Jimenez, F. Seco, C. Prieto, and J. Guevara. A comparison of Pedestrian Dead-Reckoning algorithms using a low-cost MEMS IMU. In *WISP '09*, pages 37–42.

[18] J. W. Kim, H. J. Jang, D. H. Hwang, and C. Park. A step, stride and heading determination for the pedestrian navigation system. *Journal of Global Positioning Systems*, 3(1-2).

[19] Y. Kim, H. Shin, and H. Cha. Smartphone-based wi-fi pedestrian-tracking system tolerating the rss variance problem. In *PerCom'12*.

[20] J. R. Kwapisz, G. M. Weiss, and S. A. Moore. Activity recognition using cell phone accelerometers. *SIGKDD Explor. Newsl.*, 12(2):74–82, Mar. 2011.

[21] F. Li, C. Zhao, F. Zhao, and et al. A reliable and accurate indoor localization method using phone inertial sensors. In *Ubicomp'12*.

[22] H. Lim, L. C. Kung, J. C. Hou, and H. Luo. Zero-configuration, robust indoor localization: Theory and experimentation. *Infocom '06*.

[23] D. Moore, J. Leonard, D. Rus, and S. Teller. Robust distributed network localization with noisy range measurements. In *SenSys '04*.

[24] J.-g. Park, B. Charrow, D. Curtis, J. Battat, E. Minkov, J. Hicks, S. Teller, and J. Ledlie. Growing an organic indoor location system. In *MobiSys '10*.

[25] N. B. Priyantha, H. Balakrishnan, E. Demaine, and S. Teller. Anchor-free distributed localization in sensor networks. Technical report, MIT CSail, 2003.

[26] A. Rai, K. K. Chintalapudi, V. N. Padmanabhan, and R. Sen. Zee: zero-effort crowdsourcing for indoor localization. In *Mobicom '12*.

[27] P. Robertson, M. Angermann, and M. Khider. Improving simultaneous localization and mapping for pedestrian navigation and automatic mapping of buildings by using online human-based feature labeling. In *IEEE/ION PLANS 2010*.

[28] P. Robertson, M. Angermann, and B. Krach. Simultaneous localization and mapping for pedestrians using only foot-mounted inertial sensors. In *Ubicomp '09*.

[29] J. Scarlett. Enhancing the performance of pedometers using a single accelerometer. In *Application Note, Analog Devices*, 2007.

[30] Y. Shang, W. Ruml, Y. Zhang, and M. P. J. Fromherz. Localization from mere connectivity. In *MobiHoc '03*.

[31] H. Shin, Y. Chon, and H. Cha. Unsupervised construction of indoor floor plan using smartphone. *IEEE Trans on Systems Man and Cybernetics. Part C-Applications and Reviews*, 2011.

[32] U. Steinhoff and B. Schiele. Dead reckoning from the pocket - an experimental study. In *PerCom '10*.

[33] S. Thrun, W. Burgard, and D. Fox. *Probabilistic robotics*. MIT Press, 2005.

[34] A. W. Tsui, Y. Hsiang Chuang, and H. Hua Chu. Unsupervised Learning for Solving RSS Hardware Variance Problem in WiFi Localization. *Mobile Networks and Applications*, 14:677–691, 2009.

[35] H. Wang, S. Sen, A. Elgohary, M. Farid, M. Youssef, and R. R. Choudhury. No need to war-drive: unsupervised indoor localization. In *MobiSys '12*, pages 197–210, New York, NY, USA, 2012. ACM.

[36] O. Woodman and R. Harle. Pedestrian localisation for indoor environments. In *UbiComp '08*.

[37] K. Yamanaka, M. Kanbara, and N. Yokoya. Localization of walking or running user with wearable 3d position sensor. In *ICAT '07*.

[38] Z. Yang, C. Wu, and Y. Liu. Locating in fingerprint space: wireless indoor localization with little human intervention. In *Mobicom '12*.

Real Time Network Policy Checking using Header Space Analysis

Peyman Kazemian[†*], Michael Chang[†], Hongyi Zeng[†],
George Varghese[‡], Nick McKeown[†], Scott Whyte[§]
[†]*Stanford University,* [‡]*UC San Diego & Microsoft Research,* [§]*Google Inc.*
[†]{kazemian, mchang91, hyzeng, nickm}@stanford.edu, [‡]varghese@cs.ucsd.edu, [§]swhyte@google.com

Abstract

Network state may change rapidly in response to customer demands, load conditions or configuration changes. But the network must also ensure correctness conditions such as isolating tenants from each other and from critical services. Existing policy checkers cannot verify compliance in real time because of the need to collect "state" from the entire network and the time it takes to analyze this state. SDNs provide an opportunity in this respect as they provide a logically centralized view from which every proposed change can be checked for compliance with policy. But there remains the need for a fast compliance checker.

Our paper introduces a real time policy checking tool called *NetPlumber* based on Header Space Analysis (HSA) [8]. Unlike HSA, however, NetPlumber *incrementally* checks for compliance of state changes, using a novel set of conceptual tools that maintain a dependency graph between rules. While NetPlumber is a natural fit for SDNs, its abstract intermediate form is conceptually applicable to conventional networks as well. We have tested NetPlumber on Google's SDN, the Stanford backbone and Internet 2. With NetPlumber, checking the compliance of a typical rule update against a single policy on these networks takes 50-500μs on average.

1 Introduction

Managing a network today manually is both cumbersome and error-prone. For example, network administrators must manually login to a switch to add an access-control rule blocking access to a server. In a recent survey [15], network administrators reported that configuration errors are very common in their networks.

The problem is that *several* entities can modify the forwarding rules: in addition to manual configuration, distributed protocols (e.g. OSPF, spanning tree, BGP) write entries into forwarding tables. There is no single location where all of the state is observable or controllable, leaving network administrators to use ad-hoc tools like ping and traceroute to indirectly probe the current state of the forwarding rules.

Recently, there has been growing interest in automating network control using software-defined networks (SDNs). SDN separates the control plane from the forwarding plane; a well-defined interface such as OpenFlow [11] lets the control plane write <*match, action*> rules to switches. The controller *controls* the forwarding state because it decides which rules to write to the switches; and it *observes* the forwarding state because it was the sole creator. SDNs therefore present an opportunity to automate the verification of correct forwarding behavior. This is the premise of recent work on automatic analysis of forwarding state for SDNs [8, 10, 14]. The basic idea is that if we can analyze the forwarding state—either as it is written to switches, or after it has been written—then we can check against a set of invariants/policies and catch bugs *before* or *soon after* they take place.

Our paper describes a verification tool called NetPlumber for SDNs and conventional networks. In SDNs, NetPlumber sits in line with the control plane, and observes state changes (e.g. OpenFlow messages) between the control plane and the switches (Figure 1). NetPlumber checks every event, such as installation of a new rule, removal of a rule, port or switch up and down events, against a set of policies and invariants. Upon detecting a violation, it calls a function to alert the user or block the change. In conventional networks, NetPlumber can get state change notifications through SNMP traps or by frequently polling switches. Our evaluations use a large SDN (Google WAN) and two medium sized IP networks (Internet2 and the Stanford Network).

NetPlumber can detect simple invariant violations such as loops and reachability failures. It can also check more sophisticated policies that reflect the desires of human operators such as: "Web traffic from A to B should never pass through waypoints C or D between 9am and 5pm." Our NetPlumber prototype introduces a new formal language (similar to FML [6]) to express policy checks, and is fast enough to perform real-time checks each time a controller adds a new rule. In experiments with the Stanford backbone, Google's WAN, and Internet2's backbone, NetPlumber typically verifies a rule change in less than 1ms, and a link-up or link-down event in a few seconds.

*Peyman Kazemian was an intern at Google while doing this work.

NetPlumber's speed easily exceeds the requirements for an enterprise network where configuration state changes infrequently—say once or twice per day. But in modern multi-tenant data centers, fast programmatic interfaces to the forwarding plane allow control programs to rapidly change the network configuration - perhaps thousands of times per second. For example, we may move thousands of virtual machines (VMs) to balance load, with each change requiring a tenant's virtual network to be reconfigured.

NetPlumber builds on our earlier work on Header Space Analysis (HSA) [8]. HSA models networks using a geometric model that is much easier to reason about than the vendor-specific interfaces on switches and routers. NetPlumber improves upon HSA in two ways. First, by running HSA checks incrementally, NetPlumber enables real-time checking of updates; this in turn can prevent bugs from occurring. Second, NetPlumber provides a flexible way to express and check complex policy queries without writing new ad hoc code for each policy check, as was required by HSA.

The four contributions of this paper are:

1. *NetPlumber* (section 3): NetPlumber is our real-time policy checking tool with sub-millisecond average run time per rule update.
2. *Flexible Policy Query Mechanism* (section 4): NetPlumber introduces a flexible way to express complex policy queries in an extensible, regular-expression-based language called FlowExp.
3. *Distributed NetPlumber* (section 5): We show how to scale NetPlumber to large networks using a distributed implementation.
4. *Evaluation at Scale* (section 6): We evaluate NetPlumber on three production networks, including Google's global WAN carrying inter-datacenter traffic.

2 Header Space Analysis

NetPlumber uses HSA [8] as a foundation. HSA provides a uniform, vendor-independent and protocol-agnostic model of the network using a geometric model of packet processing. A header is a point (and a flow is a region) in a $\{0,1\}^L$ space, called the *header space*, where each bit corresponds to one dimension of this space and L is an upper bound on header length (in bits). Networking boxes are modeled using a *Switch Transfer Function* T, which transforms a header h received on input port p to a set of packet headers on one or more output ports: $T : (h, p) \rightarrow \{(h_1, p_1), (h_2, p_2), ...\}$.

Each transfer function consists of an ordered set of *rules* R. A typical rule consists of a set of physical input ports, a *match* wildcard expression, and a set of actions to be performed on packets that match the wildcard expression. Examples of actions include: forward to a port, drop, rewrite, encapsulate, and decapsulate. Network topology is modeled using a *Topology Transfer Function*, Γ. If port p_{src} is connected to p_{dst} using a link, then Γ will have a rule that transfers (h, p_{src}) to (h, p_{dst}).

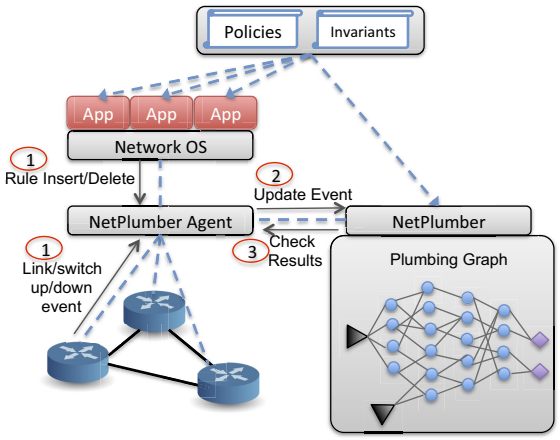


Figure 1: Deploying NetPlumber as a policy checker in SDNs.

HSA computes reachability from source A, via switches $X, Y, ...$ to destination B as follows. First, create a header space region at A representing the set of all possible packets A could send: the all-wildcard flow with L wildcard bits and covering the entire L-dimensional space. Next, apply switch X's transfer function to the all-wildcard flow to generate a set of regions at its output ports, which in turn are fed to Y's switch transfer function. The process continues until a subset of the flows that left A reach B. While the headers may have been transformed in the journey, the original headers sent by A can be recovered by applying the *inverse transfer function*. Despite considerable optimization, the Python-based implementation called Hassel described in [8] requires tens of seconds to compute reachability.

3 NetPlumber

NetPlumber is much faster than Hassel at update time because instead of recomputing all the transformations each time the network changes, it incrementally updates only the portions of those transfer function results affected by the change. Underneath, NetPlumber still uses HSA. Thus, it inherits from HSA the ability to verify a wide range of policies—including reachability between ports, loop-freedom, and isolation between groups—while remaining protocol agnostic.

Figure 1 shows NetPlumber checking policies in an SDN. An agent sits between the control plane and switches and sends every state update (installation or removal of rules, link up or down events) to NetPlumber which in turn updates its internal model of the network; if a violation occurs, NetPlumber performs a user-defined action such as removing the violating rule or notifying

the administrator.

The heart of NetPlumber is the *plumbing graph* which captures all possible paths of flows[1] through the network. Nodes in the graph correspond to the *rules* in the network and directed edges represent the *next hop dependency* of these rules:

- A rule is an OpenFlow-like `<match, action>` tuple where the action can be `forward`,[2] `rewrite`, `encapsulate`, `decapsulate`, etc.
- Rule A has a next hop dependency to rule B if 1) there is a physical link from rule A's box to rule B's box; *and* 2) the domain of rule B has an intersection with range of rule A. The domain of a rule is the set of headers that match on the rule and the range is the region created by the `action` transformation on the rule's domain.

Initialization: NetPlumber is initialized by examining the forwarding tables to build the plumbing graph. Then it computes reachability by computing the set of packets from source port s, that can reach destination port d by injecting an"all-wildcard flow" at s and propagating it along the edges of the plumbing graph. At each rule node, the flow is filtered by the `match` part of the rule and then transformed by the `action` part of the rule. The resulting flow is then propagated along the outgoing edges to the next node. The portion of the flow, if any, that reaches d is the set of all packets from s that can reach d. To speed up future calculations, whenever a rule node transforms a flow, it remembers the flow. This caching lets NetPlumber quickly update reachability results every time a rule changes.

Operation: In response to insertion or deletion of rules in switches, NetPlumber adds or removes nodes and updates the routing of flows in the plumbing graph. It also re-runs those policy checks that need to be updated.

3.1 The Plumbing Graph

The nodes of the plumbing graph are the forwarding rules, and directed edges represent the next-hop dependency of these rules. We call these directed edges *pipes* because they represent possible paths for flows. A pipe from rule a to b has a *pipe filter* which is the intersection of the range of a and the domain of b. When a flow passes through a pipe, it is filtered by the pipe filter. Conceptually the pipe filter represents all packet headers at the output of rule a that can be processed by b.

A rule node corresponds to a rule in a forwarding table in some switch. Forwarding rules have priorities; when a packet arrives to the switch it is processed by the highest priority matching rule. Similarly, the plumbing graph needs to consider rule priorities when deciding which rule node will process a flow. For computational efficiency, each rule node keeps track of higher priority rules in the same table. It calculates the domain of each higher priority rule, subtracting it from its own domain. We refer to this as *intra-table dependency* of rules.

Figure 2 shows an example network and its corresponding plumbing graph. It consists of 4 switches, each with one forwarding table. For simplicity, all packet headers are 8 bits. We will use this example though the rest of this section.

Let's briefly review how the plumbing graph of Figure 2 is created: There is a pipe from rule 1 in table 1 (rule 1.1) to rule 2 in table 2 (rule 2.2) because (a) ports 2 and 4 are connected and (b) the range of rule 1.1 (1010xxxx) and the domain of rule 2.2 (10xxxxxx) has a non-empty intersection (pipe filter: 1010xxxx). Similarly there is a pipe from rule 2.2 to rule 4.1 because (a) ports 5 and 8 are connected and (b) the range of rule 2.2 (111xxxxx) and the domain of rule 4.1 (xxxxx010) has a non-empty intersection (pipe filter: 111xx010). Also rule 1.1 has an intra-table influence on rule 1.3 because their domains and input port sets have a non-empty intersection (intersecting domain: 1010xxxx, port: 1). The rest of this plumbing graph is created in similar fashion.

3.2 Source and Sink Nodes

NetPlumber converts policy and invariants to equivalent reachability assertions. To compute reachability, it inserts flow from the source port into the plumbing graph and propagates it towards the destination. This is done using a "flow generator" or *source node*. Just like rule nodes, a source node is connected to the plumbing graph using directed edges (pipes), but instead of processing and forwarding flows, it generates flow.

Continuing our example, we compute reachability between port 1 and 10 in Figure 3 by connecting a source node, generating the all-wildcard flow, to port 1. We have also connected a special node called a *probe* node to port 10. Probe nodes will be discussed in the next section. The flow generated by the source node first reaches rules 1.1, 1.2 and 1.3. Rule 1.1 and 1.2 are not affected by any higher priority rules and don't rewrite flows. Therefore the input flow is simply forwarded to the pipes connecting them to rule 2.2 (i.e. 1010xxxx and 10001xxx flows reach rule 2.2). However rule 1.3 has an intra-table dependency to rule 1.1 and 1.2. This means that from the incoming 10xxxxxx flow, only 10xxxxxx $-$ (1010xxxx $\cup$ 10001xxx) should be processed by rule 1.3. The remainder has already been processed by higher priority rules. Rule 1.3 is a simple forward rule and will forward the flow, unchanged, to rule 3.1. However, when this flow passes through the pipe filter between rule 1.3 and

[1] In what follows, a flow corresponds to any region of header space.

[2] A drop rule is a special case of forward rule with empty set of output ports.

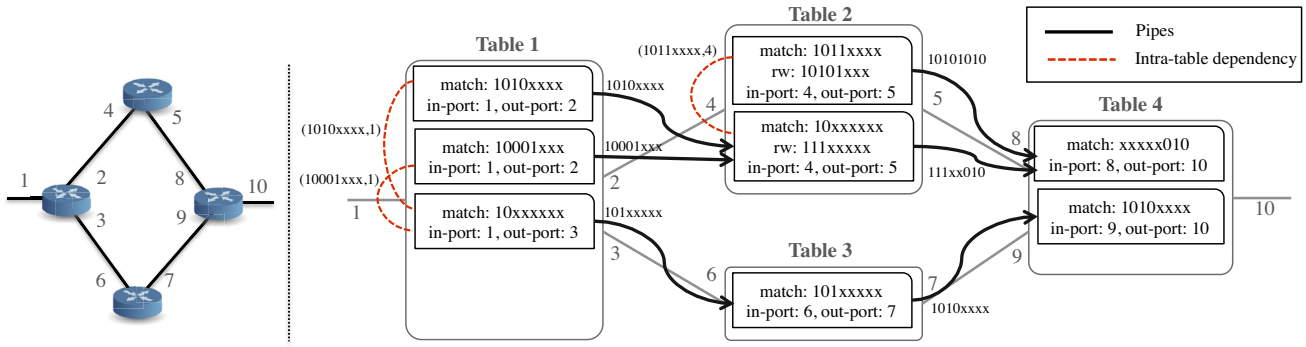


Figure 2: Plumbing graph of a simple network consisting of 4 switches each with one table. Arrows represent pipes. Pipe filters are shown on the arrows. Dashed lines indicate intra-table dependency of rules. The intersecting domain and input port is shown along the dashed lines.

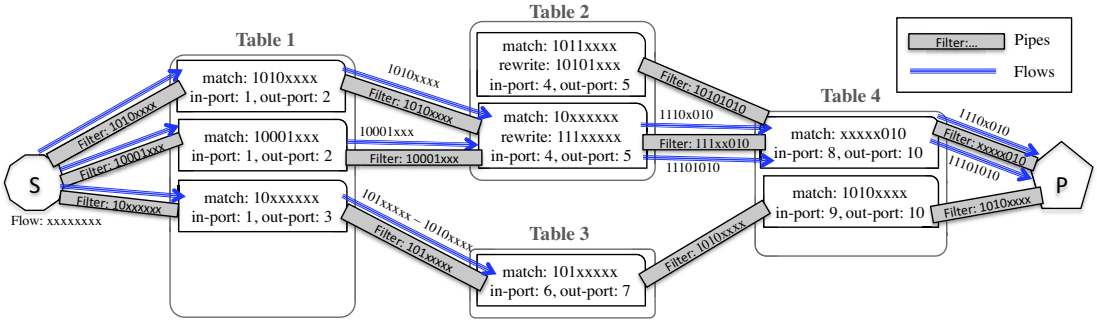


Figure 3: Finding reachability between S and P. Source node S is generating all-wildcard flow and inserting it into the plumbing graph. The solid lines show the path of flow from the source to the destination. Flow expressions are shown along the flows.

3.1 (101xxxxx), it shrinks to 101xxxxxx − 1010xxxx.[3]

The flows which reach rule 2.2 continue propagating through the plumbing graph until they reach the probe node (P), as depicted in Figure 3. However the other flow that has reached rule 3.1 does not propagate any further as it cannot pass through the pipe connecting rule 3.1 to rule 4.2. This is because the intersection of the flow (101xxxxxx − 1010xxxx = 1011xxxx) and pipe filter (1010xxxx) is empty.

Sink Nodes: Sink nodes are the dual of source nodes. A sink node absorbs flows from the network. Equivalently, a sink node generates "sink flow" which traverses the plumbing graph in the reverse direction. When reaching a rule node, a sink flow is processed by the inverse of the rule.[4] Reachability can be computed using sink nodes: if a sink node is placed at the destination port D, then the sink flow at source port S gives us the set of packet headers from S that will reach D. Sink nodes do not increase the expressive power of NetPlumber; they only simplify or optimize some policy checks (see section 4).

[3] [10xxxxxx − (1010xxxx ∪ 10001xxx)] ∩ 101xxxxx = 101xxxxx − 1010xxxx.

[4] The inverse of a rule gives us all input flows that can generate a given flow at the output of that rule [8].

3.3 Probe Nodes

A fourth type of node called a *probe node* is used to check policy or invariants. Probe nodes can be attached to appropriate locations of the plumbing graph, and can be used to check the path and header of the received flows for violations of expected behavior. In section 4, we discuss how to check a policy using a source (sink) and probe node. As a simple example, if in our toy example of Figure 2 the policy is "port 1 and 10 can only talk using packets matching xxxxx010", then we place a source node at port 1 (S), a probe node at port 10 (P) and configure P to check whether all flows received from S match xxxxx010 (Figure 3).

Probe nodes can be of two types: *source probe nodes* and *sink probe nodes*. The former check constraints on flows generated by source nodes, and the latter check flows generated by sink nodes. We refer to both as probe nodes.

3.4 Updating NetPlumber State

As events occur in the network, NetPlumber needs to update its plumbing graph and re-route the flows. There are 6 events that NetPlumber needs to handle:

Adding New Rules: When a new rule is added, NetPlumber first creates pipes from the new rule to all po-

tential next hop rules, and from all potential previous hop rules to the new rule. It also needs to find all intra-table dependencies between the new rule and other rules within the same table. In our toy example in Figure 4, a new rule is added at the 2nd position of table 1. This creates three new pipes to rules 2.1, 2.2 and the source node, and one intra-table dependency for rule 1.4.

Next, NetPlumber updates the routing of flows. To do so, it asks all the previous hop nodes to pass their flows on the *newly created* pipes. The propagation of these flows then continues normally through the plumbing graph. If the new rule has caused any intra-table dependency for lower priority rules, we need to update the flows passing through those lower priority rules by subtracting their domain intersection from the flow. Back to the example in Figure 4, after adding the new rule, the new flows highlighted in bold propagate through the network. Also, the intra-table dependency of the new rule on rule 1.4 is subtracted from the flow received by rule 1.4. This shrinks the flow to the extent that it cannot pass through the pipe connecting it to rule 3.1 (empty flow on the bottom path).

Deleting Rules: Deleting a rule causes all flows which pass through that rule to be removed from the plumbing graph. Further, if any lower priority rule has any intra-table dependency on the deleted rule, the effect should be added back to those rules. Figure 5 shows the deletion of rule 1.1 in our toy example. Note that deleting this rule causes the flow passing through rule 1.3 to propagate all the way to the probe node, because the influence of the deleted rule is now added back.

Link Up: Adding a new link to the network may cause additional pipes to be created in the plumbing graph, because more rules will now have physical connections between them (first condition for creating a pipe). The nodes on the input side of these new pipes must propagate their flows on the new pipes, and then through the plumbing graph as needed. Usually adding a new link creates a number of new pipes, making a Link Up event slower to process than a rule update.

Link Down: When a link goes down, all the pipes created on that link are deleted from the plumbing graph, which in turn removes all the flows that pass through those pipes.

Adding New Tables: When a new table (or switch) is discovered, the plumbing graph remains unchanged. Changes occur only when new rules are added to the new table.

Deleting Tables: A table is deleted from NetPlumber by deleting all the rules contained in that table.

3.5 Complexity Analysis

The complexity of NetPlumber for the addition of a single rule is $O(r + spd)$, where r is the number of entries in each table and s is the number of source (sink) nodes attached to the plumbing graph (which is roughly proportional to the number of policies we want to check), p is the number of pipes to and from the rule and d is the diameter of the network.

The run time complexity arises as follows: when a new rule is added, we need to first find intra-table dependencies. These require intersecting the `match` portion of the new rule with the `match` of all the other rules in the same table. We also need to create new pipes by doing $O(r)$ intersections of the range of the new rule with the domain of rules in the neighboring tables ($O(r)$ such rules).

Next, we need to route flows. Let us use the term *previous nodes* to denote the set of rules which have a pipe to the new rule. First, we need to route the flows at previous nodes to the new rule. There are $O(s)$ flows on each of these previous nodes because each source (sink) node that is connected to NetPlumber can add a flow. We need to pass these flows through $O(p)$ pipes to route them to the new rule. This is $O(sp)$ work. With a *linear fragmentation*[5] argument similar to [8], there will be $O(s)$ flows that will survive this transformation through the pipes [6] (and not $O(sp)$). The surviving flows will be routed in the same manner through the plumbing graph, requiring the same $O(sp)$ work at each node in the routing path. Since the maximum path length is the diameter d, the overall run time of this phase is $O(spd)$.

We also need to take care of intra-table dependencies between this rule and lower priority rules, and subtract the domain intersection from the flows received by lower priority rules. This subtraction is done lazily and is therefore much faster than flow routing; hence we ignore its contribution to overall run time.

4 Checking Policies and Invariants

A probe node monitors flows received on a set of ports. In the plumbing graph, it is attached to the output of all the rules sending out flows on those ports. Each probe node is configured with a *filter* flow expression and a *test* flow expression. A flow expression or *flowexp* for short, is a regular expression specifying a set of conditions on the path and the header of the flows. The *filter* flowexp constrains the set of flows that should be examined by the probe node, and the *test* flowexp is the constraint that

[5] This assumption states that if we have R flows at the output of a transfer function, and we apply these flow to the next hop transfer functions with R rules per transfer function, we will get cR flows at the output where $c << R$ is a constant. This assumption is based on the observation that flows are routed end-to-end in networks. They are usually aggregated, and not randomly fragmented in the core of the network.

[6] An alternate way to reach the same conclusion is as follows: the new rule, after insertion will look like any other rule in the network, and should on average have $O(s)$ flows.

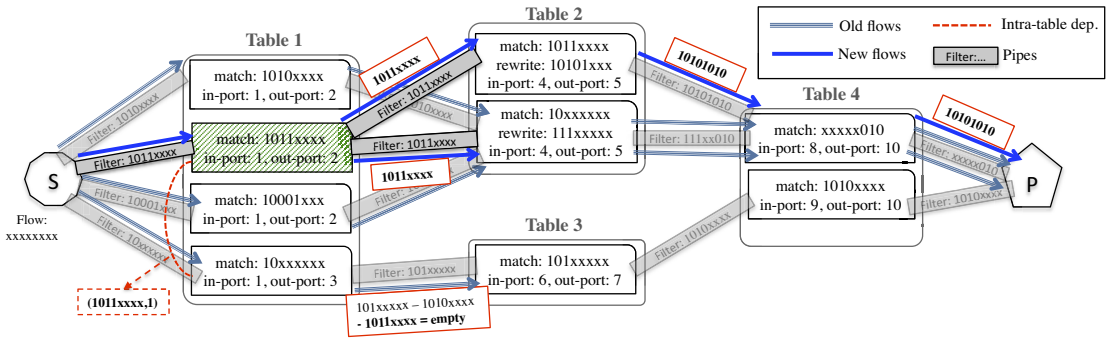


Figure 4: Adding rule 1.2 (shaded in green) to table 1. As a result a) 3 pipes are created connecting rule 1.2 to rule 2.1 and 2.2 and to the source node. b) rule 1.4 will have an intra-table dependency to the new rule (1011xxxx,1). c) The flows highlighted in bold will be added to the plumbing graph. Also the flow going out of rule 1.4 is updated to empty.

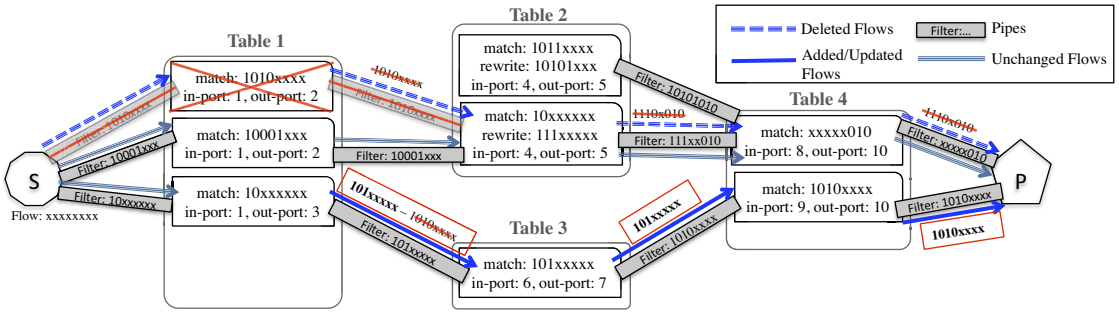


Figure 5: Deleting rule 1.1 in table 1 causes the flow which passes through it to be removed from the plumbing graph. Also since the intra-table dependency of rule 1.3 to this rule is removed, the flow passing through 1.3 through the bottom path is updated.

is checked on the matching flows. Probe nodes can be configured in two modes: *existential* and *universal*. A probe fires when its corresponding predicate is violated. An existential probe fires if none of the flows examined by the probe satisfy the test flow expression. By contrast, a universal probe fires when a single flow is received that does *not* satisfy the test constraint. More formally:

(Universal) $\forall\{f \mid f \sim filter\} : f \sim test$. All flows f which satisfy the filter expression, satisfy the test expression as well.

(Existential) $\exists\{f \mid f \sim filter\} : f \sim test$. There exist a flow f that satisfies both the filter and test expressions.

Using flow expressions described via the flowexp language, probe nodes are capable of expressing a wide range of policies and invariants. Section 4.1 will introduce the flowexp language. Sections 4.2 and 4.3 discuss techniques for checking for loops, black holes and other reachability-related policies.

4.1 Flowexp Language

Each flow at any point in the plumbing graph, carries its complete history: it has a pointer to the corresponding flow at the previous hop (node). By traversing these pointers backward, we can examine the entire history of the flow and all the rules that have processed this flow along the path. The flow history always begins at the generating source (or sink) node and ends at the probe node checking the condition.

$Constraint \rightarrow$	`True` | `False` | `!`$Constraint$
	| (*Constraint* | *Constraint*)
	| (*Constraint* & *Constraint*)
	| *PathConstraint*
	| *HeaderConstraint*;
PathConstraint $\rightarrow$	`list(` *Pathlet* `);`
Pathlet $\rightarrow$	`Port Specifier [`$p \in \{P_i\}$`]`
	| `Table Specifier [`$t \in \{T_i\}$`]`
	| `Skip Next Hop [.]`
	| `Skip Zero or More Hops [.*]`
	| `Beginning of Path [^]` `(Source/Sink node)`
	| `End of Path [$]` `(Probe node);`
HeaderConstraint $\rightarrow$	$H_{received} \cap H_{constraint} \neq \phi$
	| $H_{received} \subset H_{constraint}$
	| $H_{received} == H_{constraint}$;

Table 1: Flowexp language grammar

Flowexp is a regular expression language designed to check constraints on the history of flows received by probe nodes. Table 1 shows the grammar of flowexp in a standard BNF syntax. Flowexp consists of logical operations (i.e. *and*, *or* and *not*) on constraints enforced on the *Path* or *Header* of flows received on a probe node.

A *PathConstraint* is used to specify constraints on the

path taken by a flow. It consists of an ordered list of *pathlets* that are checked sequentially on the path of the flow. For example a flow that originates from source S, with the path $S \rightarrow A \rightarrow B \rightarrow C \rightarrow P$ to probe P, will match on flowexp "$\hat{}(p = A)$", because port A comes immediately after the source node. It also matches on "$(p = A).(p = C)$" because the flow passes through exactly one intermediate port from A to C.

A *HeaderConstraint* can check if 1) The received header has any intersection with a specified header; this is useful when we want to ensure that some packets of a specified type can reach the probe. 2) The received header is a subset of a specific header; this is useful when we wish to limit the set of headers that can reach the probe. 3) The received header is exactly equal to a specified header; this is useful to check whether the packets received at the probe are exactly what we expect.

Since flowexp is very similar to (but much simpler than) standard regular expression language, any standard regexp checking technique can be used at probe nodes.

4.2 Checking Loops and Black Holes

As flows are routed through the plumbing graph, each rule by default (i.e., without adding probe nodes for this purpose) checks received flows for loops and black holes. To check for a loop, each rule node examines the flow history to determine if the flow has passed through the current table before. If it has, a loop-detected callback function is invoked[7].

Similarly, a black hole is automatically detected when a flow is received by a non-drop-rule R that cannot pass through any pipes emanating from R. In this case, a black-hole-detected callback function is invoked.

4.3 Checking Reachability Policies

In this section, we describe how to express reachability-related policies and invariants such as the isolation of two ports, reachability between two ports, reachability via a middle box and a constraint on the maximum number of hops in a path. We express and check for such reachability constraints by attaching one or more source (or sink) nodes and one or more probe nodes in appropriate locations in the plumbing graph. The probe nodes are configured to check the appropriate filter and test flowexp constraints as shown below.

Basic Reachability Policy: Suppose we wish to ensure that a server port S should not be reachable from guest machine ports $\{G_1, ...G_k\}$.

Solution using a source probe: Place a source node that generates a wildcarded flow at each of the guest ports. Next, place a source probe node on port S and configure it to check for the flow expression: $\forall f : f.path \sim ![\hat{}(p \in \{G_1, ...G_k\})]$ - i.e., a *universal* probe with no filter constraint and a test constraint that checks that the source node in the path is not a guest port.

If, instead, the policy requires S to be reachable from $\{G_1, ...G_k\}$, we could configure the probe node as follows: $\exists f : f.path \sim [\hat{}(p \in \{G_1, ...G_k\})]$. Intuitively, this states that there exists some flow that can travel from guest ports to the server S. Note that the server S is not specified in the flow expression because the flow expression is placed at S.

Dual Solution using a sink probe: Alternately, we can put a sink node at port S and a sink probe node in each of the G_i ports. We also configure the probes with Flowexp $\forall f : f.path \sim [\hat{}(p \in \{S\})]$.

Reachability via a Waypoint: Next, suppose we wish to ensure that all traffic from port C to port S must pass through a "waypoint" node M.

Solution: Put a source node at C that generates a wildcarded flow and a probe node at S. Configure the probe node with the flow expression: $\forall\{f \mid f.path \sim [\hat{}(p \in \{C\})]\} \ : \ f.path \sim [\hat{}.*(t = M)]$. This is a universal probe which filters flows that originate from C and verifies that they pass through the waypoint M.

Path length constraint: Suppose we wish to ensure that no flow from port C to port S should go through more than 3 switches. This is a policy that was desired for the Stanford network for which we found violations. The following specification does the job assuming that each switch has one table.

Solution: Place a probe at S and a source node at C as in the previous example. Configure the probe node with the following constraint: $\forall\{f \mid f.path \sim [\hat{}(p \in \{C\})]\} \ : \ f.path \sim [\ \hat{}.\$ \mid \hat{}..\$ \mid \hat{}...\$\ ]$. The filter expression ensures that the check is done only for flows from C, and the test expression only accepts a flow if it is one, two or three hops away from the source.

Source probes versus Sink probes: Roughly speaking, if a policy is checking something at the destination regardless of where the traffic comes from, then using sink probes is more efficient. For example, suppose a manager wishes to specify that all flows arriving at a server S pass through waypoint M. Using source probes would require placing one source probe at every potential source. This can be computationally expensive as the run time of NetPlumber grows linearly with number of source or sink nodes. On the other hand, if the policy is about checking a condition for a particular source – such as computer C should be able to communicate with all other nodes – then using a source probe will be more efficient. Intuitively, we want to minimize the amount of flow in the plumbing graph required to check a given policy, as generating flow is computationally expensive.

[7]The callback function can optionally check to see if the loop is infinite or not; an algorithm to check for infinite loops is described in [8].

4.4 Policy translator

So far we have described a logical language called flowexp which is convenient for analysis and specifying precisely how flows are routed within the network. Flowexp is, however, less appropriate as a language for network managers to express higher level policy. Thus, for higher level policy specification, we decided to reuse the policy constructs proposed in the Flow-based Management Language (FML) [6], a high-level declarative language for expressing network-wide policies about a variety of different management tasks. FML essentially allows a manager to specify predicates about groups of users (e.g., faculty, students), and specifies which groups can communicate. FML also allows additional predicates on the *types* of communication allowed such as the need to pass through waypoints.

Unfortunately, the current FML implementation is tightly integrated with an OpenFlow controller, and so cannot be easily reused in NetPlumber. We worked around this by encoding a set of constructs inspired by FML in Prolog. Thus, network administrators can use Prolog as the frontend language to declare various bindings inspired by FML, such as hosts, usernames, groups and addresses. Network administrators can also use Prolog to specify different policies. For example, the following policy describes *1)* the `guest` and `server` groups, and *2)* a policy: "Traffic should go through firewall *if* it flows from a guest to a server".

```
guest(sam).
guest(michael).
server(webserver).
waypoint(HostSrc, HostDst, firewall):-
    guest(HostSrc),
    server(HostDst).
```

We have written a translator that converts such high level policy specifications written in Prolog to 1) the placement of source nodes, 2) the placement of probe nodes, and 3) the filter and test expressions for each probe node. In the example above, the translator generates two source nodes at Sam and Michael's ports and one probe node at the web server's port. The `waypoint` keyword is implemented by flowexp: *.*(t=firewall)*.

The output of the translator is, in fact, a C++ struct that lists all source, sink, and probe nodes. The source probes and sink probes are encoded in flowexp syntax using ASCII text. Finally, NetPlumber translates flowexp into C++ code that it executes.

Note that because FML is not designed to declare path constraints that can be expressed in flowexp, we found it convenient to make the translator extensible. For example, two new policy constructs we have built-in beyond the FML-inspired constructs are "at most N hops" and "immediately followed by"—but it is easy to add further constructs.

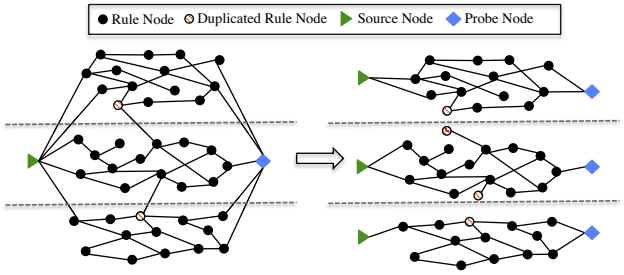


Figure 6: A typical plumbing graph consists of clusters of highly dependent rules corresponding to FECs in network. There may be rules whose dependency edges cross clusters. By replicating those rules, we can create clusters without dependencies and run each cluster as an isolated NetPlumber instance running on a different machine.

5 Distributed NetPlumber

NetPlumber is memory-intensive because it maintains considerable data about every rule and every flow in the plumbing graph. For very large networks, with millions of rules and a large number of policy constraints, NetPlumber's memory requirements can exceed that of a single machine. Further, as shown in section 3.5, the run time of NetPlumber grows linearly with the size of the tables. This can be potentially unacceptable for very large networks.

Thus, a natural approach is to run parallel instances of NetPlumber, each verifying a subset of the network and each small enough to fit into the memory of a single machine. Finally, a collector can be used to gather the check results from every NetPlumber instance and produce the final result.

One might expect to parallelize based on switches: i.e., each NetPlumber instance creates a plumbing graph for a subset of switches in the network (*vertical distribution*). This can address the memory bottleneck, but need not improve performance, as the NetPlumber instances can depend on each other. In the worst case, an instance may not be able to start its job unless the previous instance is done. This technique can also require considerable communication between different instances.

A key observation is that in every practical network we have seen, the plumbing graph looks like Figure 6: there are clusters of highly dependent rules with very few dependencies between rules in different clusters. This is caused by forwarding equivalence classes (FECs) that are routed end-to-end in the network with possible aggregation. The rules belonging to a forwarding equivalence class have a high degree of dependency among each other. For example, 10.1.0.0/16 subnet traffic might be a FEC in a network. There might be rules that further divide this FEC into smaller subnets (such as 10.1.1.0/24,

10.1.2.0/24), but there are very few rules outside this range that has any interaction with rules in this FEC (an exception is the default 0.0.0.0/0 rule).

Our distributed implementation of NetPlumber is based on this observation. Each instance of NetPlumber is responsible for checking a subset of rules that belong to one cluster (i.e. a FEC). Rules that belong to more than one cluster will be replicated on all the instances they interact with (see Figure 6). Probe nodes are replicated on all instances to ensure global verification. The final probe result is the aggregate of results generated by all the probes—i.e., all probe nodes should meet their constraints in order for the constraint to be verified. The instances do not depend on each other and can run in parallel. The final result will be ready after the last instance is done with its job.

The run time of distributed NetPlumber, running on n instances for a single rule update, is $O(m_{avg}(r/n + spd/m))$ where m is the number of times that rule get replicated and m_{avg} is the average replication factor for all rules. This is because on each replica, the size of tables are $O(m_{avg}r/n)$ and the number of pipes to a rule that is replicated m times is $O(m_{avg}p/m)$. Note that if we increase n too much, most rules will be replicated across many instances ($m, m_{avg} \rightarrow n$,) and the additional parallelism will not add any benefit.

How should we cluster rules? Graph clustering is hard in general; however for IP networks we generated natural clusters heuristically as follows. We start by creating two clusters based on the IP address of the network we are working with; if the IP address of hosts in the network belong to subnet 10.1.0.0/16, create two clusters: one for rules that match this subnet, and one for the rest (i.e. 10.1.0.0/16 and 0.0.0.0/0 - 10.1.0.0/16 subnets). Next, divide the first cluster into two clusters based on bit 17 of the destination IP address. If one of the resulting clusters is much larger than the other, we divide the larger cluster based on the next bit in IP destination address. If two clusters are roughly the same size, we divide both clusters further. This process continues until division does not reduce cluster size further (because of replication) or the specified number of clusters is reached.

Note that while we introduced the plumbing graph originally to facilitate incremental computation, the plumbing graph also allows us to decompose the computation much more effectively than the naive decomposition by physical nodes.

6 Evaluation

In this section we evaluate the performance and functionality of our C++ based implementation[8] of NetPlumber on 3 real world networks: the Google inter-datacenter WAN, Stanford's backbone network and the Internet 2 nationwide network. All the experiments are run on Ubuntu machines, with 6 cores, hyper-threaded Intel Xeon processors, a 12MB L2-cache and 12GB of DRAM.

[8]source code available at [5].

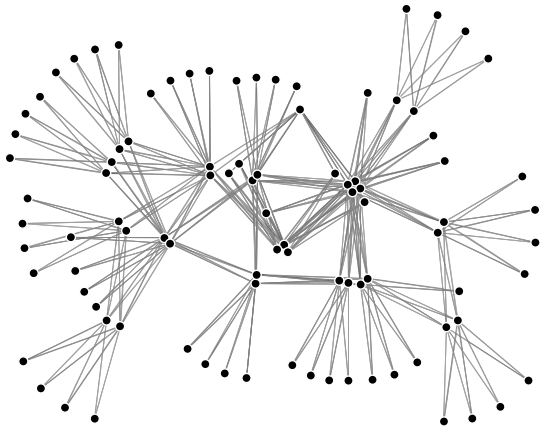


Figure 7: Google inter-datacenter WAN network.

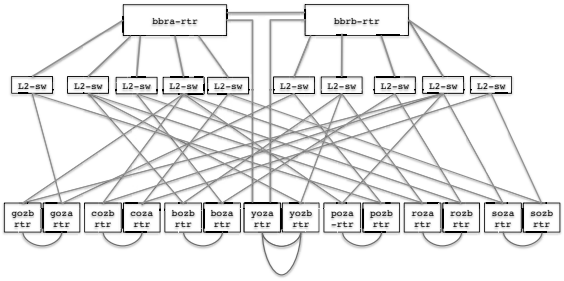


Figure 8: Stanford backbone network.

To feed the snapshot data from these networks into NetPlumber, we wrote 3 parsers capable of parsing Cisco IOS, Juniper Junos and OpenFlow dumps in protobuf [12] format. We used a json-rpc based client to feed this data into NetPlumber. NetPlumber has the json-rpc server capability and can receive and process updates from a remote source.

6.1 Our data set

Google WAN: This is a software-defined network, consisting of OpenFlow switches distributed across the globe. It connects Google data centers world-wide. Figure 7 shows the topology of this network. Overall there are more than 143,000 OpenFlow rules installed in these switches. Google WAN is one of the largest SDNs deployed today; therefore we stress-test NetPlumber on this network to evaluate its scalability.

Stanford University Backbone Network. With a population of over 15,000 students, 2,000 faculty, and five /16 IPv4 subnets, Stanford represents a mid-size enterprise network. There are 14 operational zone (OZ) Cisco routers connected via 10 Ethernet switches to 2

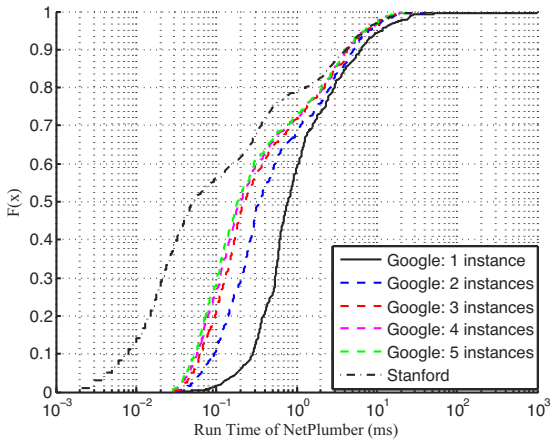


Figure 9: CDF of the run time of NetPlumber per update, when checking the all-pair reachability constraint in Google WAN with 1-5 instances and in Stanford backbone with a single instance.

#instances:	1	2	3	4	5	8
median (ms)	0.77	0.35	0.23	0.2	0.185	0.180
mean (ms)	5.74	1.81	1.52	1.44	1.39	1.32

Table 2: Average and median run time of distributed NetPlumber, checking all-pair connectivity policy on Google WAN.

backbone Cisco routers that in turn connect Stanford to the outside world (Figure 8). Overall, the network has more than 757,000 forwarding entries, 100+ VLANs and 1,500 ACL rules. Data plane configurations are collected through CLI. Stanford has made the entire configuration rule set public and it can be found in [5].

Internet2 is a nationwide backbone network with 9 Juniper T1600 routers and 100 Gb/s interfaces, supporting over 66,000 institutions in United States. There are about 100,000 IPv4 forwarding rules. All Internet2 configurations and FIBs of the core routers are publicly available [7], with the exception of ACL rules, which are removed for security reasons. We only use the IPv4 network of Internet 2 in this paper.

6.2 All-pair connectivity of Google WAN

As an internal, inter-datacenter WAN for Google, the main goal of Google WAN is to ensure connectivity between different data centers at all times. Therefore in our first experiment, we checked for the *all-pair connectivity* policy between all 52 leaf nodes (i.e. data center switches). We began by loading a snapshot of all the OpenFlow rules of Google WAN — taken at the end of July 2012 — into NetPlumber. NetPlumber created the initial plumbing graph in 33.39 seconds (an average per-rule runtime of $230\mu s$). We then attach one probe and one source node at each leaf of the network and set up the probes to look for one flow from each of the sources. If no probes fire, it means that all data centers are reachable from each other. The initial all-pair connectivity test took around 60 seconds. Note that the above run times, are for the *one-time initialization* of NetPlumber. Once NetPlumber is initialized, it can incrementally update check results much faster when changes occur. Note that the all-pair reachability check in Google WAN corresponds to 52^2 or more than 2600 pair-wise reachability checks.

Next, we used a second snapshot taken 6 weeks later. We found the diff of the two snapshots and applied them to simulate incremental updates. The diff includes both insertion and deletion of rules. Since we did not have timing information for the individual updates, we knew the *set* of updates in the difference but not the *sequence* of updates. So we simulated two different orders. In the first ordering, we applied all the rule insertions before the rule deletions. In the second ordering, we applied all deletions before all insertions.

As expected, the all-pair connectivity policy was maintained during the first ordering of update events, because new reachable paths are created before old reachable paths are removed. However the second ordering resulted in violations of the all-pair connectivity constraint *during* the rule deletion phase. Of course, this does not mean that the actual Google WAN had reachability problems because the order we simulated is unlikely to have been the actual order of updates. At the end of both orderings, the all-pair connectivity constraint was met.

NetPlumber was able to check the compliance of each insertion or deletion rule in an average time of 5.74ms with a median time of 0.77ms. The average run time is much higher than the median because there are a few rules whose insertion and deletion takes a long time (about 1 second). These are the default forwarding rules that have a large number of pipes and dependencies from/to other rules. Inserting and deleting default rules require significant changes to the plumbing graph and routing of flows. The solid line in Figure 9 shows the run time CDF for these updates.

To test the performance of distributed NetPlumber we repeated the same experiment in distributed mode. We simulated[9] the running of NetPlumber on $2-8$ machines and measured the update times (dashed lines in Figure 9). Table 2 summarizes the mean and median run times. This suggests that most of the benefits of distribution is achieved when the number of instances is 5. This is because in the plumbing graph of the Google WAN, there are about 5 groups of FECs whose rules do not influence

[9]To simulate, we run the the instances in serial on the same machine and collected the results from each run. For each rule insertion/deletion, we reported the run time as the maximum run time across all instances, because the overall job will be done only when the last instance is done.

each other. Trying to put these rules in more than 5 clusters will result in duplication of rules; the added benefit will be minimal.

6.3 Checking policy in Stanford network

Unlike the Google WAN, there are a number of reachability restrictions enforced in the Stanford network by different ACLs. Examples of such policies include isolation of machines belonging to a particular research group from the rest of the network, or limitation on the type of traffic that can be sent to a server IP address. For example, all TCP traffic to the computer science department is blocked except for those destined to particular IP addresses or TCP port numbers. In addition, there is a global reachability goal that every edge router be able to communicate to the outside world via the uplink of a specified router called `bbra_rtr`. Finally, due to the topology of the network, the network administrators desired that all paths between any two edge ports be no longer than 3 hops long to minimize network latency.

In this experiment we test all these policies. To do so, we connect 16 source nodes, one to each router in the plumbing graph. To test the maximum-3-hop constraint, we connected 14 probe nodes, one to each OZ router. We also placed a probe node at a router called `yoza_rtr` to check reachability policies at the computer science department. NetPlumber took 0.5 second to create the initial plumbing graph and 36 seconds to generate the initial check results. We found no violation of the reachability policies of the computer science department. However NetPlumber did detect a dozen un-optimized routes, whose paths take 4 hops instead of 3. We also found 10 loops, similar to the ones reported in [8][10].

We then tested the per-update run time of NetPlumber by randomly selecting 7% of rules in the Stanford network, deleting them and then adding them back. Figure 9 shows the distribution of the per-update run time. Here, the median runtime is $50\mu s$ and the mean is $2.34ms$. The huge difference between the mean and the median is due to a few outlier default rules which take a long time to get inserted and deleted into NetPlumber.

6.4 Performance benchmarking

The previous two experiments demonstrated the scalability and functionality of NetPlumber when checking actual policies and invariants of two production networks. However, the performance of NetPlumber depends on s, the number of sources in the network which is a direct consequences of the quantity and type of policies specified by each network. Thus it seems useful to have a metric that is per source node and even per policy, so we can extrapolate how run time will change as we add more independent policies, each of which require adding a new source node.[11] We provide such a unit run time benchmark for NetPlumber running on all three data sets: Google WAN, Stanford and Internet 2.

Network:	Google		Stanford		Internet 2	
Run Time	mean	median	mean	median	mean	median
Add Rule (ms)	0.28	0.23	0.2	0.065	0.53	0.52
Add Link (ms)	1510	1370	3020	2120	4760	2320

Table 3: Average and median run time of NetPlumber, for a single rule and link update, when only one source node is connected to NetPlumber.

To obtain this benchmark, we connect a single source node at one of the edge ports in the plumbing graph of each of our 3 networks. Then we load NetPlumber with 90% of the rules selected uniformly at random. Finally, we add the last 10% and measure the update time. We then repeated the same experiment by choosing links in the network that are in the path of injected flows, deleting them and then adding them back and measuring the time to incorporate the added link. The results are summarized in Table 3. As the table suggests, link up events take much longer (seconds) to incorporate. This is in fact expected and acceptable, because when a link is added, a potentially large number of pipes will be created which changes routing of flows significantly. Fortunately, since the link up/down event should be rare, this run time appears acceptable.

7 Discussion

Conventional Networks: Conceptually, NetPlumber can be used with conventional networks as long as we implement a notification mechanism for getting updated state information. One way to do this is through SNMP traps; every time a forwarding entry or link state changes, NetPlumber gets a notification. The drawback of such a mechanism is resource consumption at the switch.

Handling Transient Violations: Sometimes, during a sequence of state updates, transient policy violations may be acceptable (e.g. a black hole is acceptable while installing a path in a network). NetPlumber probes can be turned off during the transition and turned on when the update sequence is complete.

Handling Dynamic Policies: In multi-tenant data centers, the set of policies might change dynamically upon VM migration. NetPlumber can handle dynamic policy changes easily. In the plumbing graph, if we attach a source node to every edge port (as we did in the case of Google WAN), we can update policies by changing the locations and test conditions of probe nodes. This update is fast as long as the structure of the plumbing graph and routing of flows doesn't change.

[10] We used the same snapshots.

[11] By contrast, dependent policies can be checked using a single source node.

Limitations of NetPlumber: NetPlumber, like HSA relies on reading the state of network devices and therefore cannot model middleboxes with dynamic state. To handle such dynamic boxes, the notion of "flow" should be extended to include other kind of state beyond header and port. Another limitation of NetPlumber is its greater processing time for verifying link updates. As a result, it is not suitable for networks with a high rate of link up/down events such as energy-proportional networks.

8 Related Work

Recent work on network verification, especially on troubleshooting SDNs, focuses on the following directions.

Programming foundations: Frenetic [3] provides high-level abstractions to achieve per-packet and per-flow consistency during network updates [13]. NetPlumber, on the other hand, verifies forwarding policies.

Offline checking: `rcc` [2] verifies BGP configurations. NICE [1] applies model checking techniques to find bugs in OpenFlow control programs. HSA [8] checks data plane correctness against invariants. Anteater [10] uses boolean expressions and SAT solvers for network modeling and checking. However, offline checking cannot prevent bugs from damaging the network until the periodic check runs.

Online monitoring: Several tools help troubleshoot network programs at run-time. OFRewind [14] captures and reproduces the sequence of problematic OpenFlow command sequence. ATPG [16] systematically generates test packets against router configurations, and monitors network health by perioidically sending these tests packets. NDB [4] is a network debugger. These tools complement but not replace the need for real-time policy verification.

VeriFlow [9] is the work most closely related to NetPlumber. VeriFlow also verifies the compliance of network updates with specified policies in real time. It uses a *trie structure* to search rules based on equivalence classes (ECs), and upon an update, determines the affected ECs and updates the forwarding graph for that class. This in turn triggers a rechecking of affected policies. NetPlumber and VeriFlow offer similar runtime performance. While both systems support verification of forwarding actions, NetPlumber additionally can verify arbitrary header modifications, including rewriting and encapsulation. NetPlumber is also protocol-independent.

9 Conclusions

This paper introduces NetPlumber as a real-time policy checker for networks. Unlike earlier work that checks periodic snapshots of the network, NetPlumber is fast enough to validate every update in real time. Users can express a wide range of policies to be checked using an extensible regular-expression like language, called Flowexp. Since Flowexp might be too low-level for administrators to use, we implemented a higher level policy language (inspired by FML) implemented in Prolog.

The fundamental idea of the dependency graph formalized as a plumbing graph benefits us in three ways. First, it allows incremental computation by allowing only the (smaller) dependency subgraph to be traversed when a new rule is added. Second, it naturally leads us to generalize to probe nodes that can be configured to check for new policies—without the ad hoc programming effort required by Hassel. Third, clustering the graph to minimize inter-cluster edges provides a powerful way to parallelize computation.

NetPlumber is useful as a foundation that goes beyond static policy checking. For example, it can be used in ATPG [16] to allow the suite of ATPG tests packets to be updated swiftly when the configuration changes. Also NDB [4] may benefit from NetPlumber. Like GDB, NDB allows setting break points in the system when a specified condition is met. To achieve this goal, NDB adds a "postcard generating action" that captures and sends samples of matching packets to a central database. NetPlumber can be used to notify NDB when a rule that requires postcard action is about to be added to the network. While these are only two examples, we believe that the ability to incrementally and quickly do header space analysis will be a fundamental building block for network verification tools going forward.

10 Acknowledgements

We would like to thank our shepherd, Brad Karp, and the anonymous reviewers for their valuable comments. We thank Faro Rabe (Google) for providing the snapshots of Google WAN and Urs Hölzle and Stephen Stuart (Google) for internal review of this work. This research was sponsored by Google Summer Internship program, Stanford CS Undergraduate Research Internship (CURIS) program and NSF grant CNS-0855268.

References

[1] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE way to test openflow applications. In *Proceedings of NSDI'12*, 2012.

[2] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proceedings of NSDI'05*, pages 43–56, 2005.

[3] 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):279–291, Sept. 2011.

[4] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown. Where is the debugger for my software-

defined network? In *Proceedings of HotSDN '12*, pages 55–60, 2012.

[5] Header Space Library and NetPlumber. https://bitbucket.org/peymank/hassel-public/.

[6] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical declarative network management. In *Proceedings of WREN '09*, pages 1–10, 2009.

[7] The Internet2 Observatory Data Collections. http://www.internet2.edu/observatory/archive/data-collections.html.

[8] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *Proceedings of NSDI'12*, 2012.

[9] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: verifying network-wide invariants in real time. In *Proceedings of NSDI'13*, 2013.

[10] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *Proceedings of SIGCOMM '11*, pages 290–301, 2011.

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM CCR.*, 38:69–74, March 2008.

[12] Protobuf. http://code.google.com/p/protobuf/.

[13] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proceedings of SIGCOMM '12*, pages 323–334, 2012.

[14] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: enabling record and replay troubleshooting for networks. In *Proceedings of USENIX-ATC'11*, 2011.

[15] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. A Survey on Network Troubleshooting. Technical Report Stanford/TR12-HPNG-061012, Stanford University, June 2012.

[16] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. In *Proceedings of CoNEXT 2012*, Nice, France, December 2012.

Ensuring Connectivity via Data Plane Mechanisms

Junda Liu[‡], Aurojit Panda[♮], Ankit Singla[†], Brighten Godfrey[†], Michael Schapira[◇], Scott Shenker[♮♠]
[‡]Google Inc., [♮]UC Berkeley, [†]UIUC, [◇]Hebrew U., [♠]ICSI

Abstract

We typically think of network architectures as having two basic components: a data plane *responsible for forwarding packets at line-speed, and a* control plane *that instantiates the forwarding state the data plane needs. With this separation of concerns, ensuring connectivity is the responsibility of the control plane. However, the control plane typically operates at timescales several orders of magnitude slower than the data plane, which means that failure recovery will always be slow compared to data plane forwarding rates.*

In this paper we propose moving the responsibility for connectivity to the data plane. Our design, called Data-Driven Connectivity (DDC) ensures routing connectivity via data plane mechanisms. We believe this new separation of concerns — basic connectivity on the data plane, optimal paths on the control plane — will allow networks to provide a much higher degree of availability, while still providing flexible routing control.

1 Introduction

In networking, we typically make a clear distinction between the *data plane* and the *control plane*. The data plane forwards packets based on local state (*e.g.,* a router's FIB). The control plane establishes this forwarding state, either through distributed algorithms (*e.g.,* routing) or manual configuration (*e.g.,* ACLs for access control). In the naive version of this two-plane approach, the network can recover from failure only after the control plane has computed a new set of paths and installed the associated state in all routers. The disparity in timescales between packet forwarding (which can be less than a microsecond) and control plane convergence (which can be as high as hundreds of milliseconds) means that failures often lead to unacceptably long outages.

To alleviate this, the control plane is often assigned the task of precomputing failover paths; when a failure occurs, the data plane utilizes this additional state to forward packets. For instance, many datacenters use ECMP, a data plane algorithm that provides automatic failover to another shortest path. Similarly, many WAN networks use MPLS's Fast Reroute to deal with failures on the data plane. These "failover" techniques set up additional, but static, forwarding state that allows the datapath to deal with one, or a few, failures. However, these methods require careful configuration, and lack guarantees. Such configuration is tricky, requiring operators to account for complex factors like multiple link failures, and correlated failures. Despite the use of tools like shared-risk link groups to account for these issues, a variety of recent outages [21, 29, 34, 35] have been attributed to link failures. While planned backup paths are perhaps enough for most customer requirements, they are still insufficient when stringent network resilience is required.

This raises a question: can the failover approach be extended to more general failure scenarios? We say that a data plane scheme provides *ideal forwarding-connectivity* if, for any failure scenario where the network remains physically connected, its forwarding choices would guide packets to their intended destinations.[1] Our question can then be restated as: can any approach using static forwarding state provide ideal forwarding-connectivity? We have shown (see [9] for a precise statement and proof of this result) that without modifying packet headers (as in [16, 18]) the answer is *no*: one cannot achieve ideal forwarding-connectivity with static forwarding state.

Given that this impossibility result precludes ideal forwarding-connectivity using *static* forwarding information, the question is whether we can achieve ideal forwarding-connectivity using state change operations that can be executed at data plane timescales. To this end, we propose the idea of *data-driven connectivity* (DDC), which maintains forwarding-connectivity via simple changes in forwarding state predicated only on the destination address and incoming port of an arriving packet. DDC relies on state changes which are simple enough to be done at packet rates with revised hardware (and, in current routers, can be done quickly in software). Thus, DDC can be seen as moving the responsibility for connectivity to the data plane.

The advantage of the DDC paradigm is that it leaves the network functions which require global knowledge (such as optimizing routes, detecting disconnections, and distributing load) to be handled by the control plane, and moves connectivity maintenance, which has simple yet crucial semantics, to the data plane. DDC can react, at worst, at a much faster time scale than the control plane, and with new hardware can keep up with the data plane.

DDC's goal is simple: ideal connectivity with data plane mechanisms. It does not bound latency, guarantee in-order packet delivery, or address concerns of routing

[1]Note that ideal forwarding-connectivity does not guarantee packet delivery, because such a guarantee would require dealing with packet losses due to congestion and link corruption.

policy; leaving all of these issues to be addressed at higher layers where greater control can be exercised at slower timescales. Our addition of a slower, background control plane which can install arbitrary routes safely even as DDC handles data plane operations, addresses the latency and routing policy concerns over the long term.

We are unaware of any prior work towards the DDC paradigm (see discussion of related work in §5). DDC's algorithmic foundations lie in *link reversal routing* (Gafni and Bertsekas [10], and subsequent enhancements [8, 26, 32]). However, traditional link reversal algorithms are not suited to the data plane. For example, they involve generating special control packets and do not handle message loss (*e.g.,* due to physical layer corruption). In addition, our work extends an earlier workshop paper [17], but the algorithm presented here is quite different in detail, is provably correct, can handle arbitrary delays and losses, and applies to modern chassis switch designs (where intra-switch messaging between linecards may exhibit millisecond delays).

2 DDC Algorithm

2.1 System Model

We model the network as a graph. The assumptions we make on the behavior of the system are as follows.

Per-destination serialization of events at each node. Each node in the graph executes our packet forwarding (and state-update) algorithm serially for packets destined to a particular destination; there is only one such processing operation active at any time. For small switches, representing the entire switch as a single node in our graph model may satisfy this assumption. However, a single serialized node is a very unrealistic model of a large high-speed switch with several linecards, where each linecard maintains a FIB in its ASIC and processes packets independently. For such a large switch, our abstract graph model has *one node for each linecard*, running our node algorithm in parallel with other linecards, with links between all pairs of linecard-nodes within the same switch chassis. We thus only assume each linecard's ASIC executes packets with the same destination serially, which we believe is an accurate model of real switches.

Simple operations on packet time scales. Reading and writing a handful of FIB bits associated with a destination and executing a simple state machine can be performed in times comparable to several packet processing cycles. Our algorithm works with arbitrary FIB update times, but the performance during updates is sub-optimal, so we focus on the case where this period is comparable to the transmission time for a small number of packets.

In-order packet delivery along each link. This assumption is easily satisfied when switches are connected physically. For switches that are separated by other network elements, GRE (or other tunneling technologies) with sequence numbers will enforce this property. Hardware support for GRE or similar tunneling is becoming more common in modern switch hardware.

Unambiguous forwarding equivalence classes. DDC can be applied to intradomain routing at either layer 2 or layer 3. However, we assume that there is an unambiguous mapping from the "address" in the packet header to the key used in the routing table. This is true for routing on MAC addresses and MPLS labels, and even for prefix-based routing (LPM) as long as every router uses the same set of prefixes, but fails when aggregation is nonuniform (some routers aggregate two prefixes, while others do not). This latter case is problematic because a given packet will be associated with different routing keys (and thus different routing entries). MPLS allows this kind of aggregation, but makes explicit when the packet is being routed inside a larger Forwarding Equivalence Class. Thus, DDC is not universally applicable to all current deployments, but can be used by domains which are willing to either (a) use a uniform set of prefixes or (b) use MPLS to implement their aggregation rather than using nonuniform prefixes.

For convenience we refer to the keys indexing into the routing table as destinations. Since DDC's routing state is maintained and modified independently across destinations, our algorithms and proofs are presented with respect to one destination.

Arbitrary loss, delay, failures, recovery. Packets sent along a link may be delayed or lost arbitrarily (*e.g.,* due to link-layer corruption). Links and nodes may fail arbitrarily. A link or node is not considered recovered until it undergoes a control-plane recovery (an AEO operation; §3). This is consistent with typical router implementations which do not activate a data plane link until the control plane connection is established.

2.2 Link Reversal Background

DDC builds on the classic Gafni-Bertsekas (**GB**) [10] link reversal algorithms. These algorithms operate on an abstract directed graph which is at all times a directed acyclic graph (**DAG**). The goal of the algorithm is to modify the graph incrementally through **link reversal** operations in order to produce a "destination-oriented" DAG, in which the destination is the only node with no outgoing edges (i.e., a **sink**). As a result, all paths through the DAG successfully lead to the destination.

The GB algorithm proceeds as follows: Initially, the graph is set to be an arbitrary DAG. We model link directions by associating with each node a variable *direction* which maps that node's links to the data flow direction (In or Out) for that link. Throughout this section we describe our algorithms with respect to a particular destination. Sinks other than the destination are **activated** at arbitrary times; a node v, when activated, executes the following algorithm:

```
GB_activate(v)
  if for all links L, direction[L] = In
      reverse all links
```

Despite its simplicity, the GB algorithm converges to a destination-oriented DAG in finite time, regardless of the pattern of link failures and the initial link directions, as long as the graph is connected [10]. Briefly, the intuition for the algorithm's correctness is as follows. Suppose, after a period of link failures, the physical network is static. A node is *stable* at some point in time if it is done reversing. If any node is unstable, then there must exist a node u which is unstable but has a stable neighbor w. (The destination is always stable.) Since u is unstable, eventually it reverses its links. But at that point, since w is already stable, the link $u \rightarrow w$ will never be reversed, and hence u will always have an outgoing link and thus become stable. This increases the number of stable nodes, and implies that all nodes will eventually stabilize. In a stable network, since no nodes need to reverse links, the destination must be the only sink, so the DAG is destination-oriented as desired. For an illustrated example of GB algorithm execution, we refer the reader to Gafni-Bertsekas' seminal paper [10].

Gafni-Bertsekas also present (and prove correct) a *partial reversal* variant of the algorithm: Instead of reversing all links, node v keeps track of the set S of links that were reversed by its neighbors since v's last reversal. When activated, if v is a sink, it does two things: (1) It reverses $N(v) \setminus S$—unless *all* its links are in S, in which case it reverses all its links. (2) It empties S.

However, GB is infeasible as a data plane algorithm: To carry out a reversal, a node needs to generate and send special messages along each reversed link; the proofs assume these messages are delivered reliably and immediately. Such production and processing of special packets, potentially sent to a large number of neighbors, is too expensive to carry out at packet-forwarding timescales. Moreover, packets along a link may be delayed or dropped; loss of a single link reversal notification in GB can cause a permanent loop in the network.

2.3 Algorithm

DDC's goal is to implement a link reversal algorithm which is suited to the data plane. Specifically, all events are triggered by an arriving data packet, employ only simple bit manipulation operations, and result only in the forwarding of that single packet (rather than duplication or production of new packets). Moreover, the algorithm can handle arbitrary packet delays and losses.

The DDC algorithm provides, in effect, an emulation of GB using only those simple data plane operations. Somewhat surprisingly, we show this can be accomplished without special signaling, using only a *single bit* piggybacked in each data packet header—or equivalently, *zero bits*, with two virtual links per physical link. Virtual links can be implemented as GRE tunnels.

The DDC algorithm follows. Our presentation and implementation of DDC use the partial reversal variant of GB, which generally results in fewer reversals [4]. However, the design and proofs work for either variant.

State at each node:

- `to_reverse`: List containing a subset of the node's links, initialized to the node's incoming links in the given graph G.

Each node also keeps for each link `L`:

- `direction[L]`: In or Out; initialized to the direction according to the given graph G. Per name, this variable indicates this node's view of the direction of the link `L`.
- `local_seq[L]`: One-bit unsigned integer; initialized to 0. This variable is akin to a version or sequence number associated with this node's view of link `L`'s direction.
- `remote_seq[L]`: One-bit unsigned integer; initialized to 0. This variable attempts to keep track of the version or sequence number at the neighbor at the other end of link `L`.

All three of these variables can be modeled as booleans, with increments resulting in a bit-flip.

State in packets: We will use the following notation for our one bit of information in each packet:

- `packet.seq`: one-bit unsigned integer.

A comparison of an arriving packet's sequence number with `remote_seq[L]` provides information on whether the node's view of the link direction is accurate, or the link has been reversed. We note that `packet.seq` can be implemented as a bit in the packet header, or equivalently, by sending/receiving the packet on one of two virtual links. The latter method ensures that *DDC requires no packet header modification*.

Response to packet events: The following two routines handle received and locally generated packets.

```
packet p generated locally:
   update_FIB_on_departure()
   send_on_outlink(any outlink, p)

packet p received on link L:
   update_FIB_on_arrival(p, L)
   update_FIB_on_departure()
   if (direction[L] = Out)
      if (p.seq != remote_seq[L])
         send_on_outlink(L, p)
   send_on_outlink(any outlink, p)

send_on_outlink(link L, packet p)
   p.seq = local_seq[L]
   send p on L
```

These algorithms are quite simple: In general, after updating the FIB (as specified below), the packet can be sent on any outlink. There is one special case which will allow us to guarantee convergence (see proof of Thm. 2.1): if a packet was *received* on an outlink without a new sequence number, indicating that the neighbor has stale information about the direction of this link, it is "bounced back" to that neighbor.

FIB update: The following methods perform local link reversals when necessary.

```
reverse_in_to_out(L)
   direction[L] = Out
   local_seq[L]++

reverse_out_to_in(L)
   direction[L] = In
   remote_seq[L]++

update_FIB_on_arrival(packet p, link L)
   if (direction[L] = In)
      assert(p.seq == remote_seq[L])
   else if (p.seq != remote_seq[L])
      reverse_out_to_in(L)

update_FIB_on_departure()
   if there are no Out links
      if to_reverse is empty
         // 'partial' reversal impossible
         to_reverse = all links
      for all links L in to_reverse
         reverse_in_to_out(L)
      // Reset reversible links
      to_reverse = {L: direction[L] = In}
```

The above algorithms determine when news of a neighbor's link reversal has been received, and when we must locally reverse links via a *partial reversal*. For the partial reversal, `to_reverse` tracks what links were incoming at the last reversal (or at initialization). If a partial reversal is not possible (i.e., `to_reverse` is empty), all links are reversed from incoming to outgoing.

To understand how our algorithms work, note that the only exchange of state between neighbors happens through `packet.seq`, which is set to `local_seq[L]` when dispatching a packet on link `L`. Every time a node reverses an incoming link to an outgoing one, it flips `local_seq[L]`. The same operation happens to `remote_seq[L]` when an outgoing link is reversed.

The crucial step of detecting when a neighbor has reversed what a node sees as an outgoing link, is performed as the check: `packet.seq` $\stackrel{?}{=}$ `remote_seq[L]`. If, in the stream of packets being received from a particular neighbor, the sequence number changes, then the link has been reversed to an incoming link.

It is possible for a node v to receive a packet on an outgoing link for which the sequence number has not changed. This indicates that v must have previously reversed the link to outgoing from incoming, but the neighbor hasn't realized this yet (because packets are in flight, or no packets have been sent on the link since that reversal, or packets were lost on the wire). In this case, no new reversals are needed; the neighbor will eventually receive news of the reversal due to the previously-discussed special case of "bouncing back" the packet.

Response to link and node events: Links to neighbors that fail are simply removed from a node's forwarding table. A node or link that recovers is not incorporated by our data plane algorithm. This recovery occurs in the control plane, either locally at a node or as a part of the periodic global control plane process; both use the AEO operation we introduce later (§3).

FIB update delay: For simplicity our exposition has assumed that the FIB can be modified as each packet is processed. While the updates are quite simple, on some hardware it may be more convenient to decouple packet forwarding from FIB modifications.

Fortunately, DDC can allow the two `update_FIB` functions to be called after some delay, or even skipped for some packets (though the calls should still be ordered for packets from the same neighbor). From the perspective of FIB state maintenance, delaying or skipping `update_FIB_on_arrival()` is equivalent to the received packet being delayed or lost, which our model and proofs allow. Delaying or skipping `update_FIB_on_departure()` has the problem that there might be no outlinks. In this case, the packet can be sent out an inlink. Since `reverse_out_to_in()` is not called, the packet's sequence number is not incremented, and the neighbor will not interpret it as a link reversal.

Of course, delaying FIB updates delays data plane convergence, and during this period packets may temporarily loop or travel on less optimal paths. However, FIB update delay is a performance, rather than a correctness, issue; and our experiments have not shown significant problems with reasonable FIB update delays.

2.4 Correctness and Complexity

Our main results (proofs in Appendix A) show that DDC (a) provides ideal forwarding-connectivity; and (b) converges, *i.e.,* the number of reversal operations is bounded.

Theorem 2.1. *DDC guides*[2] *every packet to the destination, assuming the graph remains connected during an arbitrary sequence of failures.*

Theorem 2.2. *If after time t, the network has no failures and is connected, then regardless of the (possibly infinite)*

[2]By guide we mean that following the instructions of the forwarding state would deliver the packet to the destination, assuming no packet losses or corruption along the way.

sequence of packets transmitted after t, DDC incurs $O(n^2)$ reversal operations for a network with n nodes.

We note one step in proving the above results that may be of independent interest. Our results build on the traditional Gafni-Bertsekas (GB) link reversal algorithm, but GB is traditionally analyzed in a model in which reversal notifications are immediately delivered to all of a node's neighbors, so the DAG is always in a globally-consistent state.[3] However, we note that these requirements can be relaxed, without changing the GB algorithm.

Lemma 2.3. *Suppose a node's reversal notifications are eventually delivered to each neighbor, but after arbitrary delay, which may be different for each neighbor. Then beginning with a weakly connected DAG (i.e., a DAG where not all nodes have a path to the destination) with destination d, the GB algorithm converges in finite time to a DAG with d the only sink.*

2.5 Proactive Signaling

The data plane algorithm uses packets as reversal notifications. This implies that if node *A* reverses the link connecting it to *B*, *B* learns of the reversal only when a packet traverses the link. In some cases this can result in a packet traversing the same link twice, increasing path stretch. One could try to overcome this problem by having *A* proactively send a packet with TTL set to 1, thus notifying *B* of this change. This packet looks like a regular data packet, that gets dropped at the next hop router.

Note that proactive signaling is an entirely optional optimization. However, such signaling does not address the general problem of the data plane deviating from optimal paths to maintain connectivity. The following section addresses this problem.

3 Control Plane

While DDC's data plane guarantees ideal connectivity, we continue to rely on the *control plane* for path optimality. However, we must ensure that control plane actions are compatible with DDC's data plane. In this section, we present an algorithm to guide the data plane to use paths desired by the operator (*e.g.,* least-cost paths). The algorithm described does not operate at packet timescales, and relies on explicit signaling by the *control plane*. We start by showing how our algorithm can guide the data plane to use shortest paths. In §3.3, we show that our method is general, and can accommodate any DAG.

We assume that each node is assigned a *distance* from the destination. These could be produced, for instance, by a standard shortest path protocol or by a central coordinator. We will use the distances to define a *target* DAG on the graph by directing edges from higher- to lower-distance nodes, breaking ties arbitrarily but consistently (perhaps by comparing node IDs). Note that this target DAG may not be destination-oriented — for instance the shortest path protocol may not have converged, so distances are inconsistent with the topology. Given these assigned distances, our algorithm guarantees the following properties:

- **Safety:** Control plane actions must not break the data plane guarantees, even with arbitrary simultaneous dynamics in both data and control planes.
- **Routing Efficiency:** After the physical network and control plane distance assignments are static, if the target DAG is destination-oriented, then the data plane DAG will match it.

These guarantees are not trivial to provide. Intuitively, if the control plane modifies link directions while the data plane is independently making its own changes, it is easy for violations of safety to arise. The most obvious approach would be for a node to unilaterally reverse its edges to match the target DAG, perhaps after waiting for nodes closer to the destination to do the same. But dynamics (e.g., link failures) during this process can quickly lead to loops, so packets will loop indefinitely, violating safety. We are attempting to repair a running engine; our experience shows that even seemingly innocuous operations can lead to subtle algorithmic bugs.

3.1 Control Plane Algorithm

Algorithm idea: We decompose our goals of safety and efficiency into two modules. First, we design an all-edges-outward (**AEO**) operation which modifies all of a single node's edges to point outward, and is guaranteed not to violate safety regardless of when and how AEOs are performed. For example, a node which fails and recovers, or has a link which recovers, can unilaterally decide to execute an AEO in order to rejoin the network.

Second, we use AEO as a subroutine to incrementally guide the network towards shortest paths. Let $v_1, \ldots, v_n$ be the (non-destination) nodes sorted in order of distance from the destination, i.e., a topological sort of the target DAG. Suppose we iterate through each node i from 1 to n, performing an AEO operation on each v_i. The result will be the desired DAG, because for any undirected edge (u, v) such that u is closer to the destination, v will direct the edge $v \to u$ *after* u directed it $u \to v$.

Implementation overview: The destination initiates a heartbeat, which propagates through the network serving as a trigger for each node to perform an AEO. Nodes ensure that in applying the trigger, they do not precede neighbors who occur before them in the topological sort order. Note that if the target DAG is not destination-oriented, nodes may be triggered in arbitrary order. However, this does not affect *safety*; further, if the target DAG

[3] For example, [19] notes that the GB algorithm's correctness proof "requires tight synchronization between neighbors, to make sure the link reversal happens atomically at both ends ... There is some work required to implement this atomicity."

is destination-oriented, the data plane will converge to it, thus meeting the routing efficiency property.

We next present our algorithm's two modules: the all-edges-outward (AEO) operation (§3.1.1), and Trigger heartbeats (§3.1.2). We will assume throughout that all signals are sent through reliable protocols and yield acks/nacks. We also do not reinvent sequence numbers for heartbeats ignoring the related details.

3.1.1 All edges outward (AEO) operations

A node setting all its edges to point outward in a DAG cannot cause loops[4]. However, the data plane dynamics necessitate that we use caution with this operation—there might be packets in flight that attempt to reverse some of the edges inwards as we attempt to point the rest outwards, potentially causing loops. One could use distributed locks to pause reversals during AEO operations, but we cannot block the data plane operations as this would violate the ideal-connectivity guarantee provided by DDC.

Below is an algorithm to perform an AEO operation at a node *v* safely and without pausing the data plane, using virtual nodes (vnodes). We use virtual nodes as a convenient means of versioning a node's state, to ensure that the DAG along which a packet is forwarded remains consistent. The strategy is to connect a new vnode *vn* to neighbors with all of *vn*'s edges outgoing, and then delete the old vnode. If any reversals are detected during this process, we treat the process as failed, delete *vn*, and continue using the old vnode at *v*. Bear in mind that this is all with respect to a specific destination; *v* has other, independent vnodes for other destinations. Additionally, although the algorithm is easiest to understand in terms of virtual nodes, it can be implemented simply with a few extra bits per link for each destination[5].

We require that neighboring nodes not perform control plane reversals simultaneously. This is enforced by a simple lock acquisition protocol between neighbors before performing other actions in AEO. However, note that these locks only pause other control-plane AEO operations; all data plane operations remain active.

```
AEO algorithm:
   Get locks from {neighbors, self} in
                increasing ID order
   Create virtual node vn
   run(thread_watch_for_packets)
   run(thread_connect_virtual_node)

thread_watch_for_packets:
   if a data packet arrives at vn
      kill thread_connect_virtual_node
      delete vn
      exit thread_watch_for_packets

thread_connect_virtual_node:
   For each neighbor u of v
      Link vn, u with virtual link
      Signal(LinkDone?) to u

   After all neighbors ack(LinkDone):
      For each neighbor u of v
         Signal(Dir: vn->u?) to u
      After all neighbors ack(Dir: vn->u):
         kill thread_watch_for_packets
         delete old virtual nodes at v
   exit thread_connect_virtual_node

When all threads complete:
    release all locks
```

The algorithm uses two threads, one to watch for data packets directed to the destination (which would mean the neighbor has reversed the link), and the other to establish links with the neighbors directed towards them, using signal-ack mechanisms. The second set of signals and acks might appear curious at first, but it merely confirms that no reversals were performed before *all* acks for the *first* set had been *received* at *vn*. There may have been reversals since any of the second set of acks were dispatched from the corresponding neighbor, but they are inconsequential (as our proof will show).

3.1.2 Trigger heartbeats

We make use of periodic heartbeats issued by the destination to trigger AEO operations. To order these operations, a node uses heartbeat H as a trigger after making sure all of its neighbors with lower distances have already responded to the H. More specifically, a node, *v*, responds to heartbeat H from neighbor *w* as follows:

```
If (last_heartbeat_processed >= H)
   Exit

rcvd[H,w] = true
If (rcvd[H,u] = true for all nbrs u with
                     lower distance)
   If (direction[u] = In for any nbr with
                     lower distance)
       AEO(v)
   Send H to all neighbors
   last_heartbeat_processed = H
```

With regard to the correctness of this algorithm, we prove the following theorem in Appendix B:

Theorem 3.1. *The control plane algorithm satisfies the safety and routing efficiency properties.*

3.2 Physical Implementation

A vnode is merely an abstraction containing the state described in §2.3, and allowing this state to be modified in the ways described previously. One can therefore

[4] This is why we designed the algorithm as a sequence of AEOs.

[5] Routers implement ECMP similarly: In a *k*-port router, *k*-way ECMP [7] stores state items per (destination, output-port) pair.

represent a vnode as additional state in a switch, and interactions with other switches can be realized using virtual links. As stated previously, the virtual links themselves may be implemented using GRE tunnels, or by the inclusion of additional bits in the packet header.

3.3 Control Plane Generality

It is easy to show that the resulting DAG from the AEO algorithm is entirely determined by the order in which AEO operations are carried out. While trigger heartbeats as described in §3.1.2 order these AEO operations by distance, any destination-oriented DAG could in fact be used. It is also easy to see that given a DAG, one can calculate at least one order of AEO operations resulting in the DAG.

Given these observations, the control plane described above is entirely general allowing for the installation of an arbitrary DAG, by which we imply that given another control plane algorithm, for which the resultant routes form a DAG, there exists a modified trigger heartbeat function that would provide the same functionality as the given control plane algorithm. DDC therefore does not preclude the use of other control plane algorithms that might optimize for metrics other than path length.

3.4 Disconnection

Detecting disconnection is particularly important for link-reversal algorithms. If our data plane algorithm fails to detect that a destination is unreachable, packets for that destination might keep cycling in the connected component of the network. Packets generated for the destination may continue to be added, while none of these packets are being removed from the system. This increases congestion, and can interfere with packets for other destinations.

Since DDC can be implemented at both the network and link-layer we cannot rely on existing TTL/hop-count fields, since they are absent from most extant link-layer protocols. Furthermore, failures might result in paths that are longer than would be allowed by the network protocol, and thus TTL-related packet drops cannot be used to determine network connectivity.

Conveniently, we can use heartbeats to detect disconnection. Any node that does not receive a heartbeat from the destination within a fixed time period can assume that the destination is unreachable. The timeout period can be set to many times the heartbeat interval, so that the loss of a few heartbeats is not interpreted as disconnection.

3.5 Edge Priorities

The algorithm as specified allows for the use of an arbitrary output link (*i.e.,* packets can be sent out any output link). One can exploit this choice to achieve greater efficiency, in particular the choice of output links can be driven by a control plane specified priority. Priorities can be chosen to optimize for various objective functions, for instance traffic engineering. Such priorities can also be used to achieve faster convergence, and lower stretches, especially when few links have failed.

Edge priorities are not required for the correct functioning of DDC, and are only useful as a mechanism to increase efficiency, and as a tool for traffic engineering. Priorities can be set by the control plane with no synchronization (since they do not affect DDC's correctness), and can either be set periodically based on some global computation, or manually based on operator preference.

4 Evaluation

We evaluated DDC using a variety of microbenchmarks, and an NS-3 [23] based macrobenchmark.

4.1 Experimental Setup

We implemented DDC as a routing algorithm in NS-3. (The code is available on request.) Our implementation includes the basic data plane operations described in §2, and support for assigning priorities to ports (*i.e.,* links, which appear as ports for individual switches), allowing the data plane to discriminate between several available output ports. We currently set these priorities to minimize path lengths. We also implemented the control plane algorithm (§3), and use it to initialize routing tables. While our control plane supports the use of arbitrary DAGs, we evaluated only shortest-path DAGs.

We evaluated DDC on 11 topologies: 8 AS topologies from RocketFuel [30] varying in size from 83 nodes and 272 links (AS1221), to 453 nodes and 1999 links (AS2914); and 3 datacenter topologies—a 3-tier hierarchical topology recommended by Cisco [3], a Fat-Tree [1], and a VL2 [12] topology. However, we present only a representative sample of results here.

Most of our experiments use a link capacity of 10 Gbps. Nodes use output queuing, with drop-tail queues with a 150 KB capacity. We test both TCP (NS-3's TCP-NewReno implementation) and UDP traffic sources.

4.2 Microbenchmarks

4.2.1 Path Stretch

Stretch is defined as the ratio between the length of the path a packet takes through the network, and the shortest path between the packet's source and destination in the current network state, *i.e.,* after accounting for failures.

Stretch is affected by the topology, the number of failed links, and the choice of source and destination. To measure stretch, we selected a random source and destination pair, and failed a link on the connecting path. We then sent out a series of packets, one at a time (*i.e.,* making sure there is no more than one packet in the network at any time) to avoid any congestion drops, and observed the

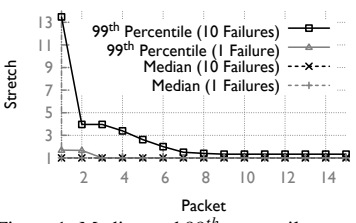


Figure 1: Median and 99^{th} percentile stretch for AS1239.

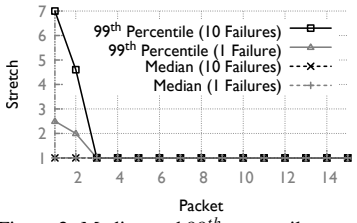


Figure 2: Median and 99^{th} percentile stretch for a FatTree

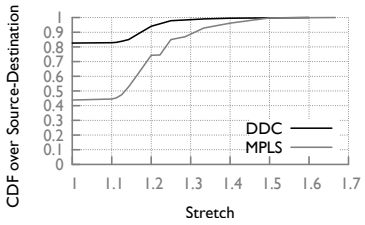


Figure 3: CDF of steady state stretch for MPLS FRR and DDC in AS2914.

length of the path the packet takes. Subsequent packets use different paths as DDC routed around the failures.

Figures 1 and 2 show stretch for a series of such packets, either with 1 or 10 links failed. We tested a wider range of link failures, but these graphs are representative of the results. As expected, the initial stretch is dependent on the number of links failed, for instance the 99^{th} percentile stretch for AS1239 with 10 link failures is 14. However, paths used rapidly converge to near-optimal.

We compare DDC's steady-state stretch with that of MPLS link protection [25]. Link protection, as commonly deployed in wide-area networks, assigns a backup path around a single link, oblivious of the destination[6]. We use the stretch-optimal strategy for link protection: the backup path for a link is the shortest path connecting its two ends. Figure 3 shows this comparison for AS2914. Clearly, path elongation is lower for DDC. We also note that link protection does not support multiple failures.

4.2.2 Packet Latency

In addition to path lengths, DDC may also impact packet latency by increasing queuing at certain links as it moves packets away from failures. While end-to-end congestion control will eventually relieve such queuing, we measure the temporary effect by comparing the time taken to deliver a packet before and after a failure.

To measure packet latencies, we used 10 random source nodes sending 1GB of data each to a set of randomly chosen destinations. The flows were rate limited (since we were using UDP) to ensure that no link was used at anything higher than 50% of its capacity, with the majority of links being utilized at a much lower capacity. For experiments with AS topologies, we set the propagation delay to 10ms, to match the order of magnitude for a wide area network, while for datacenter topologies, we adjusted propagation delay such that RTTs were $\sim$250μs, in line with previously reported measurements [5, 36].

For each source destination pair we measure baseline latency as an average over 100 packets. We then measure the latency after failing a set of links. Figure 4 shows the results for AS2914, and indicates that over 80% of packets encounter no increase in latency, and independent of the number of failures, over 96% of packets encounter only a modest increase in latency. Similarly, Figure 5 shows the same result for a Fat Tree topology, and shows that over 95% of the packets see no increased latency. In the 2 failure case, over 99% of packets are unaffected.

4.2.3 TCP Throughput and FIB Update Delay

Ideally, switches would execute DDC's small state updates at line rate. However, this may not always be feasible, so we measure the effects of delayed state updates. Specifically, we measure the effect of additional delay in FIB updates on TCP throughput in wide-area networks.

We simulated a set of WAN topologies with 1 Gbps links (for ease of simulation). For each test we picked a set of 10 source-destination pairs, and started 10 GB flows between them. Half-a-second into the TCP transfer, we failed between 1 and 5 links (the half-a-second duration was picked so as to allow TCP congestion windows to converge to their steady state), and measured overall TCP throughput. Our results are shown in Figure 6, and indicate that FIB delay has no impact on TCP throughput.

4.3 Macrobenchmarks

We also simulated DDC's operation in a datacenter, using a fat-tree topology with 8-port switches. To model failures, we used data on the time it takes for datacenter networks to react to link failures from Gill et al [11]. Since most existing datacenters do not use any link protection scheme, relying instead on ECMP and the plurality of paths available, we use a similar multipath routing algorithm as our baseline.

For our workload, we used partition-aggregate as previously described in DCTCP [2]. This workload consists of a set of background flows, whose size and interarrival frequencies we get from the original paper, and a set of smaller, latency sensitive, request queries. The request queries proceed by having a single machine send a set of 8 machines a single small request packet, and then receiving a 2 KB response in return. This pattern commonly occurs in front-end datacenters, and a set of such requests are used to assemble a single page. We generated a set of such requests, and focused on the percentage of these

[6]Protecting links in this manner is the standard method used in wide-area networks, for instance [6], states "High Scalability Solution—The Fast Reroute feature uses the highest degree of scalability by supporting the mapping of all primary tunnels that traverse a link onto a single backup tunnel. This capability bounds the growth of backup tunnels to the number of links in the backbone rather than the number of TE tunnels that run across the backbone."

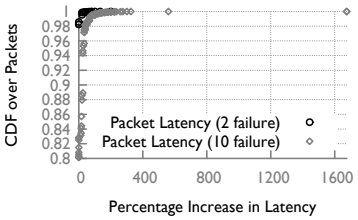


Figure 4: Packet latencies in AS2914 with link failures.

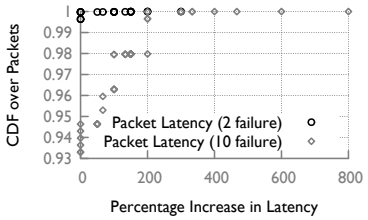


Figure 5: Packet latencies in a datacenter with a Fat-Tree topology with link failures.

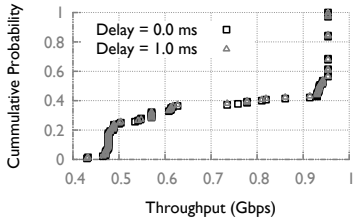


Figure 6: Distribution of TCP throughputs for varying FIB update delays.

that were satisfied during a "failure event", *i.e.,* a period of time where one or more links had failed, but before the network had reacted. On average, we generated 10 requests per second, spread evenly across 128 hosts. The hosts involved in processing a request were also picked at random. We looked at a variety of failure scenarios, where we failed between 1 and 5 links. Here, we present results from two scenarios, one where a single link was failed, and one where 5 links were failed. The links were chosen at random from a total of 384 links.

Figure 7 shows the percentage of requests served in every 10 second interval (*i.e.,* percent of request packets resulting in a response) in a case with 5 link failures. The red vertical line at $x = 50$ seconds indicates the point at which the links failed. While this is a rare failure scenario, we observe that, without DDC, in the worst case about 14% of requests cannot be fulfilled. We also look at a more common case in Figure 8 where a single link is failed, and observe a similar response rate. The response rate itself is a function of both the set of links failed, and random requests issued.

For many datacenter applications, response latency is important. Figure 9 shows the distribution of response latencies for the 5 link failure case described previously. About 4% of the requests see no responses when DDC is not used. When DDC is used, all requests result in responses and fewer than 1% see higher latency than the common case without DDC. For these 1% (which would otherwise be dropped), the latency is at most 1.5× higher. Therefore, in this environment, DDC not only delivers all requests, it delivers them relatively quickly.

5 Related Work

Link-Reversal Algorithms: There is a substantial literature on link-reversal algorithms [10, 8, 26, 32]. We borrow the basic idea of link reversal algorithms, but have extended them in ways as described in §2.2.

DAG-based Multipath: More recently there has been a mini-surge in DAG-based research [15, 28, 27, 14, 24]. All these proposals shared the general goal of maximizing robustness while guaranteeing loop-freeness. In most cases, the optimization boils down to a careful ordering of the nodes to produce an appropriate DAG. Some of this research also looked at load distribution. Our approach differs in that we don't optimize the DAG itself but instead construct a DAG that performs adequately well under normal conditions and rely on the rapid link reversal process to restore connectivity when needed.

Other Resilience Mechanisms: We have already mentioned several current practices that provide some degree of data plane resilience: ECMP and MPLS Fast Reroute. We note that the ability to install an arbitrary DAG provides strictly more flexibility than is provided by ECMP.

MPLS Fast Reroute, as commonly deployed, is used to protect individual links by providing a backup path that can route traffic around a specific link failure. Planned backups are inherently hard to configure, especially for multiple link failures, which as past outages indicate, may occur due to physical proximity of affected links, or other reasons [20]. While this correlation is often accounted for (*e.g.,* using shared risk link groups), such accounting is inherently imprecise. This is evidenced by the Internet outage in Pakistan in 2011 [21] which was caused by a failure in both a link and its backup, and other similar incidents [29, 35, 34] which have continued to plague providers. Even if *ideal connectivity* isn't an explicit goal, using DDC frees operators from the difficulties of careful backup configuration. However, if operators do have preferred backup configurations, DDC makes it possible to achieve the best of both worlds: Operators can install a MPLS/GRE tunnel (*i.e.,* a virtual link) for each desired backup path, and run DDC over the physical and virtual links. In such a deployment, DDC would only handle failures beyond the planned backups.

End-to-End Multipath: There is also a growing literature on end-to-end multipath routing algorithms (see [31] and [22] for two such examples). Such approaches require end-to-end path failure detection (rather than hop-by-hop link failure detection as in DDC), and thus the recovery time is quite long compared to packet transmission times. In addition, these approaches do not provide ideal failure recovery, in that they only compute a limited number of alternate paths, and if they all fail then they rely on the control plane for recovery.

Other Approaches: Packet Recycling [18] is perhaps the work closest in spirit to DDC (but quite different in approach), where connectivity is ensured by a packet forwarding algorithm which involves updating a logarithmic number of bits in the packet header. While this approach is a theoretical tour-de-force, it requires solving an NP-

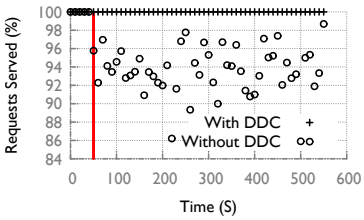


Figure 7: Percentage of requests satisfied per 10 second interval with 5 failed links.

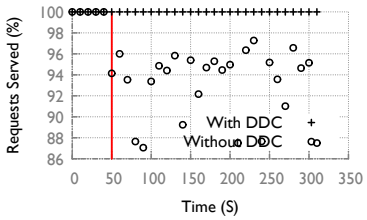


Figure 8: Percentage of requests satisfied per 10 second interval with 1 failed link

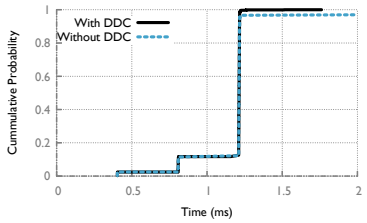


Figure 9: Request latency for the 5-link failure case in Figure 7.

hard problem to create the original forwarding state. In contract, DDC requires little in the way of precomputation, and uses only two bits in the packet header. Failure-Carrying Packets (FCP) [16] also achieves ideal connectivity, but data packets carry explicit control information (the location of failures) and routing tables are recomputed upon a packet's arrival (which may take far longer than a single packet arrival). Furthermore, FCP packet headers can be arbitrarily large, since packets potentially need to carry an unbounded amount of information about failures encountered along the path traversed.

6 Conclusion

In this paper we have presented *DDC*, a dataplane algorithm guaranteeing ideal connectivity. We have both presented proofs for our guarantees, and have demonstrated the benefits of DDC using a set of simulations. We have also implemented the DDC dataplane in OpenVSwitch[7], and have tested our implementation using Mininet [13]. We are also working towards implementing DDC on physical switches.

7 Acknowledgments

We are grateful to Shivaram Venkatraman, various reviewers, and our shepherd Dejan Kostic for their comments and suggestions. This research was supported in part by NSF CNS 1117161 and NSF CNS 1017069. Michael Schapira is supported by a grant from the Israel Science Foundation (ISF) and by the Marie Curie Career Integration Grant (CIG).

A DDC Algorithm Correctness

Compared with traditional link reversal algorithms, DDC has two challenges. First, notifications of link reversals might be delayed arbitrarily. Second, the mechanism with which we provide notifications is extremely limited—piggybacking on individual data packets which may themselves be delayed and lost arbitrarily. We deal with these two challenges one at a time.

A.1 Link reversal with delayed notification

Before analyzing our core algorithm, we prove a useful lemma: the classic GB algorithms (§2.2) work correctly even when reversal notifications are delayed arbitrarily.

[7] Available at `https://bitbucket.org/apanda/ovs-ddc`

The subtlety in this analysis is that if notifications are not delivered instantaneously, then nodes have inconsistent views of the link directions. What we will show is that when it matters—when a node is reversing its links—the node's view is consistent with a certain canonical global state that we define.

We can reason about this conveniently by defining a global notion of the graph at time t as follows: In G_t the direction of an edge (u,v) is:

- $u \to v$ if u has reversed more recently than v;
- $v \to u$ if v has reversed more recently than u;
- otherwise, whatever it is in the original graph G_0.

It is useful to keep in mind several things about this definition. First, reversing is now a local operation at a node. Once a node decides to reverse, it is "officially" reversed—regardless of when control messages are delivered to its neighbors. Second, the definition doesn't explicitly handle the case where there is a tie in the times that u and v have reversed. But this case will never occur; it is easy to see that regardless of notifications begin delayed, two neighbors will never reverse simultaneously because at least one will believe it has an outgoing edge.

Often, nodes' view of their link directions will be inconsistent with the canonical graph G_t because they haven't yet received reversal notifications. The following lemma shows this inconsistency is benign.

Lemma A.1. *Consider the GB algorithm with arbitrarily delayed reversal notifications, but no lost notifications. If v reverses at t, then v's local view of its neighboring edge directions is consistent with G_t.*

Proof. The lemma clearly holds for $t = 0$ due to the algorithm's initialization. Consider any reversal at $t \geq 0$. By induction, at the time of v's previous reversal (or at $t = 0$ if it had none), v's view was consistent with G. The only events between then and time t are edge reversals making v's edges incoming. Before v receives all reversal notifications, it believes it has an outgoing edge, and therefore will not reverse. Once it receives all notifications (and may reverse), its view is consistent with G. □

It should be clear given the above lemma that nodes only take action when they have a "correct" view of their edge directions, and therefore delay does not alter GB's effective behavior. To formalize this, we define a **trace** of an algorithm as a chronological record of its link reversals. An algorithm may have many possible traces, due to

non-determinism in when nodes are activated and when they send and receive messages (and due to the unknown data packet inputs and adversarial packet loss, which we disallow here but will introduce later).

Lemma A.2. *Any trace of GB with arbitrarily delayed notification is also a trace of GB with instant notification.*

Proof. Consider any trace T produced by GB with delay. Create a trace T' of GB with instantaneous notification, in which nodes are initialized to the same DAG as in T and nodes are activated at the moments in which those nodes reverse edges in T. We claim T and T' are identical. This is clearly true at $t = 0$. Consider by induction any $t > 0$ at which v reverses in T. In the GB algorithm, notice that v's reversal action (i.e., which subset of edges v reverses) is a function only of its neighboring edge-states at time t and at the previous moment that v reversed. By Lemma A.1, local knowledge of these edge-states are identical to G_t, which is in turn identical at each step in both T and T' (by induction). □

LEMMA 2.3. *Suppose a node's reversal notifications are eventually delivered to each neighbor, but after arbitrary delay, which may be different for each neighbor. Then beginning with a weakly connected DAG with destination d, the GB algorithm converges in finite time to a DAG with d the only sink.*

Proof. Lemmas A.1, A.2 imply that if reversal notifications are eventually delivered, both versions of the algorithm (with arbitrary delay and with instant notifications) reverse edges identically. Convergence of the algorithm with arbitrary delay to a destination-oriented DAG thus follows from the original GB algorithm's proof. □

A.2 DDC

Having shown that GB handles reversal message delay, we now show that DDC, even with its packet-triggered, lossy, delayed messages, effectively emulates GB.

Lemma A.3. *Any trace of DDC, assuming arbitrary loss and delay but in-order delivery, is a prefix of a trace of GB with instantaneous reliable notification.*

Proof. Consider any neighboring nodes v, w and any time t_0 such that:

1. v and w agree on the link direction;
2. v's `local_seq` for the link is equal to w's `remote_seq`, and vice versa; and
3. any in-flight packet p has p.`seq` set to the same value as its sender's current `local_seq`.

Suppose w.l.o.g v reverses first, at some time t_1. No packet outstanding at time t_0 or sent during $[t_0, t_1)$ can be interpreted as a reversal on delivery, since until that time, the last two properties and the in-order delivery assumption imply that such an arriving packet will have its p.`seq` equal to the receiving node's `remote_seq`. Now we have two cases. In the first case, no packets sent $v \to w$ after time t_1 are ever delivered; in this case, w clearly never believes it has received a link reversal. In the second case, some such packet is delivered to w at some time t_2. The first such packet p will be interpreted as a reversal because it will have p.`seq` $\neq w$.`remote_seq`. Note that neither node reverses the link during (t_0, t_2) since both believe they have an outlink and, like GB, DDC will only reverse a node with no outlinks.

Note that the above three properties are satisfied at time $t_0 = 0$ by initialization; and the same properties are again true at time t_2. Therefore we can iterate the argument across the entire run of the algorithm. The discussion above implies that for any a, b such that v reverses at time a and does not reverse during $[a, b]$, w will receive either zero or one reversal notifications during $[a, b]$. This is exactly equivalent to the notification behavior of GB with arbitrary delay, except that some notifications may never be delivered. Combined with the fact that DDC makes identical reversal decisions as GB when presented with the same link state information, this implies that a trace of DDC over some time interval $[0, T]$ is a *prefix* of a valid trace for GB with arbitrary delay, in which the unsent notifications are delayed until after T. Since by Lemma 2.3 any trace of GB with delay is a valid trace of GB without delay, this proves the lemma. □

While it looks promising, the lemma above speaks only of (data plane) control events, leaving open the possibility that data packets might loop forever. Moreover, since the DDC trace is only a prefix of a GB trace, the network might never converge. Indeed, if no data packets are ever sent, *nothing* happens, and even if some are sent, some parts of the network might never converge. What we need to show is that from the perspective of any data packets, the network operates correctly. We now prove the main theorems stated previously in §2.4.

THEOREM 2.1. *DDC guides every packet to the destination, assuming the graph remains connected during an arbitrary sequence of failures.*

Proof. Consider a packet p which is not dropped due to physical layer loss or congestion. At each node p is forwarded to some next node by DDC, so it must either eventually reach the destination, or travel an infinite number of hops. We will suppose it travels an infinite number of hops, and arrive at a contradiction.

If p travels an infinite number of hops then there is some node v which p visits an infinite number of times. Between each visit, p travels in a loop. We want to show that during each loop, at least one control event—either a reversal or a reversal notification delivery—happens somewhere in the network.

We show this by contradiction. If there are no control events, the global abstract graph representing the network (§A.1) is constant; call this graph G_t. By Lemma A.3, G_t must match some graph produced by GB (in the instantaneous reliable notification setting), which never has loops. Thus G_t does not have loops, so there must exist

some edge (w,u) in the packet's loop which w believes is outgoing, but which is incoming according to G_t. If this occurs, then DDC specifies that u will bounce it back to w. Therefore, by the time the packet returns to w, by the in-order delivery assumption, w will have received a reversal notification. Therefore the assumption that there are no control events must be false: some control event must happen during each loop.

Therefore, there are an infinite number of control events. Therefore, DDC has an infinitely long trace of control events, which by Lemma A.3 means it is also an infinitely-long valid prefix of a trace for GB, which contradicts the fact [10] that GB converges in a finite number of steps. Thus, the assumption that the packet is not delivered must be false, and all packets not dropped due to physical layer loss or congestion are delivered. □

THEOREM 2.2. *If after time t, the network has no failures and is connected, then regardless of the (possibly infinite) sequence of packets transmitted after t, DDC incurs $O(n^2)$ reversal operations for a network with n nodes.*

Proof. Following the same argument as above, since any DDC-trace is a prefix of a GB-trace, and the GB algorithm incurs $\Theta(n^2)$ reversals [33], DDC can not incur more reversals. □

B Control Plane Correctness

In this appendix, we prove Theorem 3.1.

Lemma B.1. *If the AEO algorithm at node v terminates with the addition of a virtual node vn, then at the time of addition of the last edge between vn and a neighbor, all other edges at vn are directed outward.*

Proof. If `thread_watch_for_packets` did not delete *vn*, it saw no incoming packets for the destination at any of *vn*'s links until each neighbor had acknowledged the direction of the link as outward from *vn*. Such packets may be in flight and some link may have been reversed *after* dispatching the ack. However, these acks are requested only after each neighbor has acknowledged that its link with *vn* is functional. Thus, between the addition of the last edge between *vn* and a neighbor's dispatching the ack for the link, all edges were directed outward. □

Lemma B.2. *Given a DAG G, an AEO operation at node v leaves it a DAG with the same physical connectivity.*

Proof. First, we note that AEO operations include obtaining a lock from all neighbors, so no two neighbors can perform an AEO operation concurrently.

AEO either terminates with the addition of vnode *vn* and deletion of the old vnode at *v*, or it terminates leaving the old vnode's *physical* connectivity unchanged. In the latter case, *vn* has not sent any packets on its links, nor has it received any packets until the first reversal that reaches it – thus, its behavior so far is identical to absence. Further, *vn* is immediately killed on receipt of a data packet, in behavior identical to a link failing after a reversal was dispatched on it. Neither scenario can leave the rest of the graph with a loop. On the other hand, if *vn* was added to the graph successfully, then by Lemma B.1, it is added as a node with all edges directed outward, again resulting in no loops. Further, in either scenario, the graph's physical connectivity remains the same – in one case, *vn* copies the old vnode's physical connectivity, while in the other, the old vnode retains connectivity as is. Thus the graph remains a DAG with the same physical connectivity. □

Given that the algorithm only performs a sequence of AEO operations, Lemma B.2 suffices to prove safety. We next show routing efficiency, which completes the proof of Theorem 3.1.

Lemma B.3. *Assume that after some point in time there are no further failures, stable distances are assigned to the nodes such that they induce a destination-oriented target DAG, and control plane messages are eventually delivered. Then the data plane DAG eventually matches the target DAG induced by the distances.*

Proof. Call a node *v compliant* when every edge outgoing from *v* in the target DAG is also outgoing from *v* in the data plane DAG. We will construct a set D of nodes which are (1) compliant, and (2) form a destination-oriented subgraph of the data plane DAG. Specifically, we will prove by induction that eventually D expands to include all nodes. Note that the definition of D immediately implies that these nodes will never reverse their links in the data plane, or execute an AEO in the control plane: they are *done*.

In the base case, D simply includes the destination. In the general case, suppose not all nodes are in D. Since the subgraph D complies with the target DAG, it also forms a destination-oriented subgraph of the target DAG. Combined with the fact that the target DAG is itself destination-oriented, this means D forms a "sink region" into which all paths in the target DAG must flow. Then if not all nodes are in D, there must exist some node $v \notin D$ which is non-compliant, but whose neighbors with lower distances are all in D. Let L be that subset of v's neighbors. Eventually, every node in L will send a heartbeat to v, and v will execute an AEO (if its edges are not already pointing outward to L). At this point, (1) v is compliant, and (2) all v's links to nodes in the destination-oriented subgraph D are outgoing, so that $D \cup \{v\}$ is itself a destination-oriented subgraph of the data plane DAG. Hence, v satisfies the two conditions for inclusion into D and the set D expands. Iterating this argument, D eventually includes all nodes, which implies the lemma. □

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[2] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). *ACM SIGCOMM Computer Communication Review*, 40(4):63–74, 2010.

[3] M. Arregoces and M. Portolani. *Data center fundamentals*. Cisco Press, 2003.

[4] B. Charron-Bost, J. Welch, and J. Widder. Link reversal: How to play better to work less. In S. Dolev, editor, *Algorithmic Aspects of Wireless Sensor Networks*, volume 5804 of *Lecture Notes in Computer Science*, pages 88–101. Springer Berlin / Heidelberg, 2009.

[5] Y. Chen, R. Griffith, J. Liu, R. Katz, and A. Joseph. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 73–82. ACM, 2009.

[6] Cisco. Mpls traffic engineering fast reroute – link protection. `http://www.cisco.com/en/US/docs/ios/12_0st/12_0st10/feature/guide/fastrout.html`.

[7] I. Cisco Systems. Cisco Nexus 3064-X and 3064-T Switches. Datasheet: `http://goo.gl/E3Wqm`, 2012.

[8] M. S. Corson and A. Ephremides. A distributed routing algorithm for mobile wireless networks. *Wireless Networks*, 1(1):61–81, 1995.

[9] J. Feigenbaum, P. B. Godfrey, A. Panda, M. Schapira, S. Shenker, and A. Singla. On the resilience of routing tables. In *Brief announcement, 31st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, July 2012.

[10] E. M. Gafni and D. P. Bertsekas. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *IEEE Transactions on Communications*, 1981.

[11] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *ACM SIGCOMM Computer Communication Review*, pages 350–361. ACM, 2011.

[12] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[13] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container based emulation. *Proc. CoNEXT (to appear)*, 2012.

[14] A. Kvalbein, A. Hansen, T. Cicic, S. Gjessing, and O. Lysne. Fast IP network recovery using multiple routing configurations. In *INFOCOM 2006*, pages 1–11. IEEE, 2007.

[15] K. Kwong, L. Gao, R. Guérin, and Z. Zhang. On the feasibility and efficacy of protection routing in IP networks. In *INFOCOM, 2010*, pages 1–9. IEEE, 2010.

[16] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica. Achieving convergence-free routing using failure-carrying packets. In *SIGCOMM*, 2007.

[17] J. Liu, B. Yang, S. Shenker, and M. Schapira. Data-driven network connectivity. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets '11, pages 8:1–8:6, New York, NY, USA, 2011. ACM.

[18] S. Lor, R. Landa, and M. Rio. Packet re-cycling: eliminating packet losses due to network failures. In *HotNets IX*, page 2. ACM, 2010.

[19] N. Lynch. Link-reversal algorithms. In *MIT 6.895 lecture notes*, April 2006. `http://courses.csail.mit.edu/6.885/spring06/notes/lect14a.pdf`.

[20] D. Madory. The 10 Most Bizarre and Annoying Causes of Fiber Cuts. Retrieved September 18, 2012: `http://goo.gl/tIATg`, 2011.

[21] D. Madory. Renesys blog: Large Outage in Pakistan. Retrieved September 18, 2012: `http://www.renesys.com/blog/2011/10/large-outage-in-pakistan.shtml`, 2011.

[22] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. SPAIN: COTS data-center ethernet for multipathing over arbitrary topologies. In *Proc. Networked Systems Design and Implementation*, Apr. 2010.

[23] ns-3. `http://www.nsnam.org/`.

[24] Y. Ohara, S. Imahori, and R. V. Meter. Mara: Maximum alternative routing algorithm. In *INFOCOM*, pages 298–306. IEEE, 2009.

[25] P. Pan, G. Swallow, and A. Atlas. RFC 4090 Fast Reroute Extensions to RSVP-TE for LSP Tunnels, May 2005.

[26] V. Park and M. Corson. A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks. In *INFOCOM*, 1997.

[27] S. Ray, R. Guérin, K. Kwong, and R. Sofia. Always acyclic distributed path computation. *IEEE/ACM Transactions on Networking (ToN)*, 18(1):307–319, 2010.

[28] C. Reichert, Y. Glickmann, and T. Magedanz. Two routing algorithms for failure protection in IP networks. In *Computers and Communications, 2005. ISCC 2005. Proceedings. 10th IEEE Symposium on*, pages 97–102. IEEE, 2005.

[29] R. Singel. Threat Level: Fiber Optic Cable Cuts Isolate Millions From Internet. Retrieved September 18, 2012: `http://www.wired.com/threatlevel/2008/01/fiber-optic-cab/`, 2008.

[30] N. Spring, R. Mahajan, and D. Wetherall. Measuring isp topologies with rocketfuel. In *In Proc. ACM SIGCOMM*, pages 133–145, 2002.

[31] M. Suchara, D. Xu, R. Doverspike, D. Johnson, and J. Rexford. Network architecture for joint failure recovery and traffic engineering. In *Proc. ACM SIGMETRICS*, June 2011.

[32] J. Welch and J. Walter. Link reversal algorithms. *Synthesis Lectures on Distributed Computing Theory*, 2(3):1–103, 2011.

[33] J. L. Welch and J. E. Walter. Link reversal algorithms. *Synthesis Lectures on Distributed Computing Theory*, 2(3):1–103, 2012/01/27 2011.

[34] Wikitech: Site issue Aug 6 2012. Retrieved September 18, 2012: `http://wikitech.wikimedia.org/view/Site_issue_Aug_6_2012`, 2012.

[35] C. Wilson. 'Dual' fiber cut causes Sprint outage. Retrieved September 18, 2012: `http://connectedplanetonline.com/access/news/Sprint_service_outage_011006/`, 2006.

[36] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: Meeting deadlines in datacenter networks. *SIGCOMM-Computer Communication Review*, 41(4):50, 2011.

Juggling the Jigsaw: Towards Automated Problem Inference from Network Trouble Tickets

Rahul Potharaju* Purdue University
Navendu Jain Microsoft Research
Cristina Nita-Rotaru Purdue University

Abstract

This paper presents NetSieve, a system that aims to do automated problem inference from network trouble tickets. Network trouble tickets are diaries comprising fixed fields and free-form text written by operators to document the steps while troubleshooting a problem. Unfortunately, while tickets carry valuable information for network management, analyzing them to do problem inference is extremely difficult—fixed fields are often inaccurate or incomplete, and the free-form text is mostly written in natural language.

This paper takes a *practical* step towards automatically analyzing natural language text in network tickets to infer the problem symptoms, troubleshooting activities and resolution actions. Our system, NetSieve, combines statistical natural language processing (NLP), knowledge representation, and ontology modeling to achieve these goals. To cope with ambiguity in free-form text, NetSieve leverages learning from human guidance to improve its inference accuracy. We evaluate NetSieve on 10K+ tickets from a large cloud provider, and compare its accuracy using (a) an expert review, (b) a study with operators, and (c) vendor data that tracks device replacement and repairs. Our results show that NetSieve achieves 89%-100% accuracy and its inference output is useful to learn global problem trends. We have used NetSieve in several key network operations: analyzing device failure trends, understanding why network redundancy fails, and identifying device problem symptoms.

1 Introduction

Network failures are a significant contributor to system downtime and service unavailability [12, 13, 47]. To track network troubleshooting and maintenance, operators typically deploy a trouble ticket system which logs all the steps from opening a ticket (e.g., customer complaint, SNMP alarm) till its resolution [21]. Trouble tickets comprise two types of fields: (a) structured data often generated automatically by alarm systems such as ticket id, time of alert, and syslog error, and (b) free-form text written by operators to record the diagnosis steps and communication (e.g., via IM, email) with the customer or other technicians while mitigating the problem. Even though the free-form field is less regular and precise compared to the fixed text, it usually provides a *detailed* view of the problem: what happened? what troubleshooting was done? and what was the resolution? Figure 1 shows a ticket describing continuous reboots of a load balancer even after reseating its power supply units; bad memory as the root cause; and memory replacement as the fix; which would be hard to infer from coarse-grained fixed data.

STRUCTURED

Ticket Title:	Ticket #xxxxxx NetDevice; LoadBalancer Down 100% Summary: Indicates that the root cause is a failed system		
Problem Type	**Problem SubType**	**Priority**	**Created**
Severity - 2	2: Medium		

UNSTRUCTURED (Diary)

Operator 1: Both power supplies have been reseated
Operator 1: The device has been powered back up and it does not appear that it has come back online. Please advise.
Operator 2: Ok. Let me see what I can do.
--- Original Message ---
From: Vendor Support
Subject: Regarding Case Number #yyyyyy
Title: **Device** v9.4.5 **continuously rebooting**
As discussed, the **device** has **bad memory chips** as such we **replace** it. Please completely fill the **RMA** form below and **return** it.

Figure 1: An example network trouble ticket.

Unfortunately, while tickets contain valuable information to infer problem trends and improve network management, mining them automatically is extremely hard. On one hand, the fixed fields are often inaccurate or incomplete [36]. Our analysis (§2.1) on a large ticket dataset shows that the designated problem type and subtype fields had incorrect or inconclusive information in 69% and 75% of the tickets, respectively. On the other hand, since the free-form text is written in natural language, it is often ambiguous and contains typos, grammatical errors, and words (e.g., "cable", "line card", "power supply") having domain-specific meanings different from the dictionary.

Given these fundamental challenges, it becomes difficult to automatically extract *meaning* from raw ticket text even with advanced NLP techniques, which are designed

*Work done during an internship at Microsoft Research, Redmond.

Table 1: Examples of network trouble tickets and their inference output from NetSieve.

		Inference output from NetSieve		
	Ticket Title	**Problems**	**Activities**	**Actions**
1	SNMPTrap LogAlert 100%: Internal link 4.8 is unavailable.	link down, failover, bad sectors	swap cable, upgrade fiber, run fsck, verify HDD	replace cable, HDD
2	HSRPEndpoint SwitchOver 100%: The status of HSRP endpoint has changed since last polling.	firmware error, interface failure	verify and break-fix supervisor engine	replace supervisor engine, reboot switch
3	StandbyFail: Failover condition, this standby will not be able to go active.	unexpected reboot, performance degraded	verify load balancer, run config script	rma power supply unit
4	The machine can no longer reach internet resources. Gateway is set to load balancer float IP.	verify static route	reboot server, invoke failover, packet capture	rehome server, reboot top-of-rack switch
5	Device console is generating a lot of log messages and not authenticating users to login.	sync error, no redundancy	power down device, verify maintenance	replace load balancer
6	Kernel panic 100%: CPU context corrupt.	load balancer reboot, firmware bug	check performance, break-fix upgrade	upgrade BIOS, reboot load balancer
7	Content Delivery Network: Load balancer is in bad state, failing majority of keep-alive requests.	standby dead, misconfigured route	upgrade devices	replace standby and active, deploy hot-fix
8	OSPFNeighborRelationship Down 100%: This OSPF link between neighboring endpoints is down.	connectivity failure, packet errors	verify for known maintenance	replace network card
9	HighErrorRate: Summary: http://domain/characteristics.cgi?<device>.	packet errors	verify interface	cable and xenpak module replaced
10	AllComponentsDown: Summary: Indicates that all components in the redundancy group are down;	down alerts	verify for decommissioned devices	decommission load balancer

to process well-written text (e.g., news articles) [33]. Most prior work on mining trouble tickets use either keyword search and manual processing of free-form content [20, 27, 42], predefined rule set from ticket history [37], or document clustering based on manual keyword selection [36]. While these approaches are simple to implement and can help narrow down the types of problems to examine, they risk (1) inaccuracy as they consider only the presence of a keyword regardless of where it appears (e.g., "do not replace the cable" specifies a negation) and its relationship to other words (e.g., "checking for maintenance" does not clarify whether the ticket was *actually* due to maintenance), (2) a significant human effort to build the keyword list and repeating the process for new tickets, and (3) inflexibility due to predefined rule sets as they do not cover unexpected incidents or become outdated as the network evolves.

Our Contributions. This paper presents NetSieve, a problem inference system that aims to automatically analyze ticket text written in natural language to infer the problem symptoms, troubleshooting activities, and resolution actions. Since it is nearly impractical to understand any arbitrary text, NetSieve adopts a domain-specific approach to first build a knowledge base using existing tickets, automatically to the extent possible, and then use it to do problem inference. While a ticket may contain multiple pieces of useful information, NetSieve focuses on inferring three key features for summarization as shown in Table 1:

1. **Problems** denote the network entity (e.g., router, link, power supply unit) and its associated state, condition or symptoms (e.g., crash, defective, reboot) as identified by an operator e.g., bad memory, line card failure, crash of a load balancer.
2. **Activities** indicate the steps performed on the network entity during troubleshooting e.g., clean and swap cables, verify hard disk drive, run configuration script.
3. **Actions** represent the resolution action(s) performed on the network entity to mitigate the problem e.g., upgrade BIOS, rehome servers, reseat power supply.

To achieve this functionality, NetSieve combines techniques from several areas in a novel way to perform problem inference over three phases. First, it constructs a domain-specific knowledge base and an ontology model to interpret the free-form text using pattern mining and statistical NLP. In particular, it finds important domain-specific words and phrases (e.g., "supervisor engine", "kernel", "configuration") and then maps them onto the ontology model to specify relationships between them. Second, it applies this knowledge base to infer problems, activities and actions from tickets and exports the inference output for summarization and trend analysis. Third, to improve the inference accuracy, NetSieve performs incremental learning to incorporate human feedback.

Our evaluation on 10K+ network tickets from a large cloud provider shows that NetSieve performs automated problem inference with 89%-100% accuracy, and several network teams in that cloud provider have used its inference output to learn global problem trends: (1) compare device reliability across platforms and vendors, (2) analyze cases when network redundancy failover is ineffective, and (3) prioritize checking for the top-k problems

and failing components during network troubleshooting.

This paper makes the following contributions:

- A large-scale measurement study (§2) to highlight the challenges in analyzing structured data and free-form text in network trouble tickets.
- Design and implementation (§3) of NetSieve, an automated inference system that analyzes free-form text in tickets to extract the problem symptoms, troubleshooting activities and resolution actions.
- Evaluation (§4) of NetSieve using expert review, study with network operators and vendor data, and showing its applicability (§5) to improve network management.

Scope and Limitations: NetSieve is based on analyzing free-form text written by operators. Thus, its accuracy is dependent on (a) fidelity of the operators' input and (b) tickets containing sufficient information for inference. NetSieve leverages NLP techniques, and hence is subject to their well-known limitations such as ambiguities caused by *anaphoras* (*e.g.*, referring to a router as *this*), complex negations (*e.g.*, "device gets replaced" but later in the ticket, the action is negated by the use of an anaphora) and truth conditions (*e.g.*, "please replace the unit once you get more in stock" does not clarify whether the unit has been replaced). NetSieve inference rules may be specific to our ticket data and may not apply to other networks. While we cannot establish representativeness, this concern is alleviated to some extent by the size and diversity of our dataset. Finally, our ontology model represents one way of building a knowledge base, based on discussions with operators. Given that the ticket system is subjective and domain-specific, alternative approaches may work better for other systems.

2 Measurement and Challenges

In this section, we present a measurement study to highlight the key challenges in automated problem inference from network tickets. The dataset comprises 10K+ (absolute counts omitted due to confidentiality reasons) network tickets logged during April 2010-2012 from a large cloud provider. Next, we describe the challenges in analyzing fixed fields and free-form text in trouble tickets.

2.1 Challenges: Analyzing Fixed Fields

C1: Coarse granularity. The fixed fields in tickets contain attributes such as 'ProblemType' and 'ProblemSubType', which are either pre-populated by alarm systems or filled in by operators. Figure 2 shows the top-10 problem types and sub-types along-with the fraction of

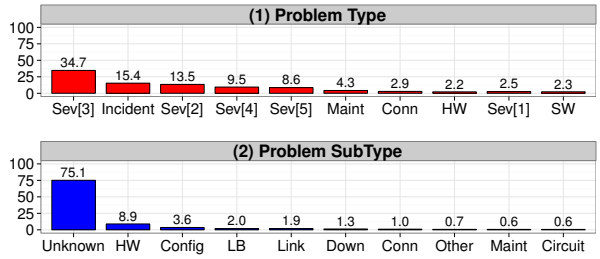


Figure 2: Problem types and subtypes listed in the tickets.

tickets. *Sev* denotes problem severity assigned based on SLAs with the customer. We observe that while problem types such as *Software*, *Hardware*, *Maintenance*, and *Incident* provide coarse granularity information about the problem type, other types e.g., *Sev[1-5]* are highly subjective reflecting operator's judgement and they account for 68.8% of the tickets. As a result, these fields are not useful to precisely infer the observed problems.

C2: Inaccuracy or Incompleteness. Figure 3 shows the problem categorization for a randomly selected subset of tickets labeled by a domain expert (top) and the field values from the tickets (bottom) for three different types of devices: (1) Access Routers (AR), (2) Firewalls, and (3) Load balancers (LB); the number of tickets is 300, 42, and 299, respectively.

We make two key observations. First, the Problem SubType field (in the bottom row) is *Unknown* in about 79%-87% of the tickets. As a result, we may incorrectly infer that devices failed due to unknown problems, whereas the problems were precisely reported in the expert labeled set based on the same ticket data. Second, the categories annotated by the expert and ticket fields for each device type have little overlap, and even when there is a common category, there is a significant difference in the fraction of tickets attributed to that category e.g., 'Cable' accounts for 0.6% of the LB tickets whereas the ground truth shows their contribution to be 9.7%.

The reason that these fields are inaccurate or incomplete is that operators work under a tight time schedule, and they usually have a narrow focus of mitigating a problem rather than analyzing failure trends. Thus, they may not have the time, may not be motivated, or simply forget to input precise data for these fields after closing the tickets. Further, some fixed fields have a drop-down menu of pre-defined labels and every problem may not be easily described using them.

2.2 Challenges: Analyzing Free-form Text

In comparison to structured data, the free-form text in network tickets is descriptive and ambiguous: it has

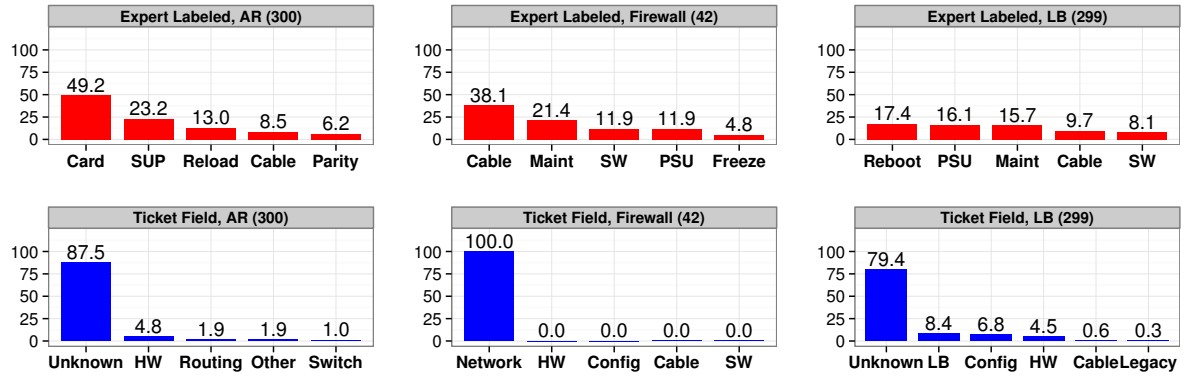


Figure 3: Categorization of the 'Problem SubType' field in tickets for (a) Access Routers (AR), (b) Firewalls, and (c) Load balancers (LB). The top and bottom rows show the major problem subtypes as labeled by an expert and the ticket field, respectively.

domain-specific words and synonyms mixed with regular dictionary words, spelling and grammar errors, and writings from different operators.

Specifically, we highlight the following challenges in mining free-form text in trouble tickets:

C1: Diversity of content. A ticket may contain a variety of semantic elements such as emails, IMs, device debug logs, devices names, and operator notes.

C2: Domain-specific words. Without a prior list of domain-specific keywords, training spell checkers can be hard e.g., DMZ and DNS are both valid technical keywords, but they cannot be found in the dictionary.

C3: Redundant text. Tickets often contain text fragments that appear with high frequency. We observe three types of frequently occurring fragments (see Figure 1): templates, emails and device logs. Templates are text fragments added by operators to meet triage guidelines, but they often do not contain any problem-specific information. Many emails are asynchronous replies to a previous message and thus, it may be hard to reconstruct the message order for inference. Log messages are usually appended to a ticket in progress. Therefore, text mining using metrics such as term frequency may incorrectly give more weightage to terms that appear in these logs.

Overall, these challenges highlight the difficulty in automatically inferring problems from tickets. While we studied only our ticket dataset, our conversation with operators (having a broader industry view and some having worked at other networks), suggests that these challenges are similar to those of many other systems.

3 Design and Implementation

In this section, we first give an overview of NetSieve and then describe its design and implementation.

3.1 Design Goals

To automatically analyze free-form text, NetSieve should meet the following design goals:

1. *Accuracy*: The inference system needs to be accurate as incorrect inference can lead to bad operator decisions, and wasted time and effort in validating inference output for each ticket, thus limiting practicality.
2. *Automation*: Although we cannot completely eliminate humans from the loop, the system should be able to operate as autonomously as possible.
3. *Adaptation*: As the network evolves, the system should be able to analyze new types of problems and leverage human feedback to acquire new knowledge for continuously improving the inference accuracy.
4. *Scalability*: The system should be scalable to process a large number of tickets where each ticket may comprise up to a million characters, in a reasonable time.
5. *Usability*: The output from the inference system should provide a user-friendly interface (e.g., visualization, REST, plaintext) to allow the operator to browse, filter and process the inference output.

3.2 Overview

NetSieve infers three key features from network trouble tickets: (1) Problem symptoms indicating what problem occurred, (2) Troubleshooting activities describing the diagnostic steps, and (3) Resolution actions denoting the fix applied to mitigate the problem.

Figure 4 shows an overview of the NetSieve architecture. NetSieve operates in three phases. First, the knowledge building phase constructs a domain-specific knowledge base and an ontology model using existing tickets and input from a domain-expert. Second, the operational

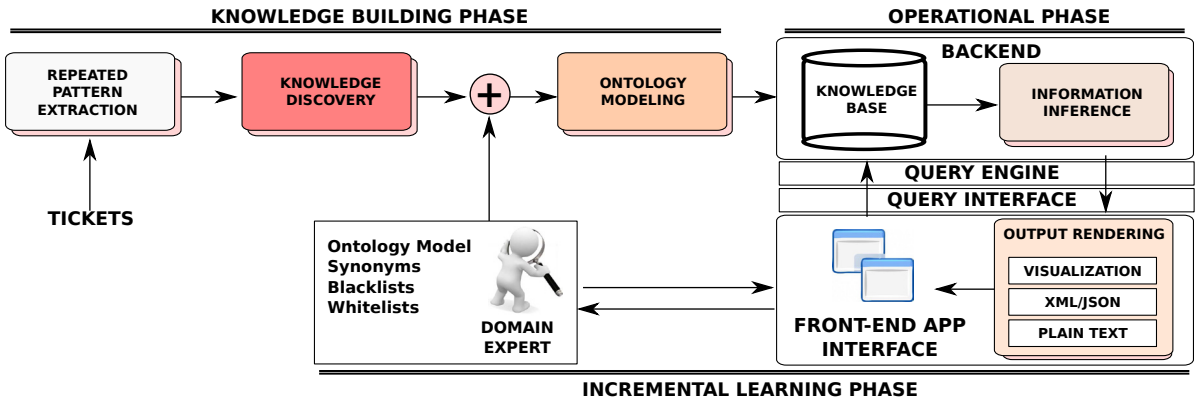


Figure 4: NetSieve Architecture: The first phase builds a domain-specific knowledge base using existing tickets. The second phase uses the knowledge base to make problem inference. The third phase leverages human guidance to improve the inference accuracy.

phase uses the knowledge base to make problem inference from tickets. Third, the incremental learning phase improves the accuracy of knowledge base using human guidance. We next give a brief description of each of these phases.

Knowledge Building Phase: The goal of this phase is to analyze free-form text to extract important domain-specific phrases such as "power supply unit" and "load balancer" using repeated pattern mining (§3.3.1) and statistical NLP (§3.3.2). These domain-specific phrases are then mapped onto an ontology model (§3.3.3) that formally represents the relationships between network entities and stores them in a knowledge base. This phase is executed either when NetSieve is bootstrapped or to retrain the system using expert feedback.

Operational Phase: The goal of this phase is to perform problem inference (§3.4) from a ticket using the knowledge base. To export the inference output, NetSieve supports SQL (through the *Query Engine*) and HTTP GET requests (through a *Query Interface* such as REST [11]) and outputs results in a variety of data formats such as XML/JSON, and through data visualization for ticket summarization and trend analysis.

Incremental Learning Phase: To improve inference accuracy, it is important to continuously update the knowledge base to incorporate any new domain-specific terminologies. NetSieve provides an interface to allow a domain-expert to give feedback for improving the ontology model, synonyms, blacklists and whitelists. After each learning session, NetSieve performs problem inference using the updated knowledge base.

3.3 Knowledge Building Phase

Building a domain-specific knowledge phase requires addressing three key questions. First, what type of information should be extracted from the free-form text to enable problem inference? Second, how do we extract this information in a scalable manner from a large ticket corpus? Third, how do we model the relationships in the extracted information to infer *meaning* from the ticket content. Next we describe solutions to these questions.

3.3.1 Repeated Pattern Extraction

Intuitively, the phrases that would be most useful to build a knowledge base should capture domain-specific information and be related to *hot* (common) and important problem types. As mining arbitrary ticket text is extremely hard (§2), we first extract hot phrases and later apply filters (§3.3.2) to select the important ones.

DESIGN: To find the hot phrases from ticket text, we initially applied conventional text mining techniques for *n*-gram extraction. *N*-grams are arbitrary and recurrent word combinations [4] that are repeated in a given context [45]. Since network tickets have no inherent linguistic model, we extracted n-grams of arbitrary length for comprehensive analysis without limiting to bi-grams or tri-grams. We implemented several advanced techniques [9, 39, 45] from computational linguistics and NLP, and observed the following challenges:

1. Extracting all possible n-grams can be computationally expensive for a large n and is heavily dependent on the size of the corpus. We investigated using a popular technique by Nagao et al. [39] based on extracting word co-locations, implemented in C [56]. On our dataset, this algorithm did not terminate on a 100K word document after 36 hours of CPU time on a Xeon 2.67 GHz eight-core server with 48 GB RAM, as also observed by others [52].
2. Determining and fine-tuning the numerous thresholds and parameters used by statistical techniques [39, 45,

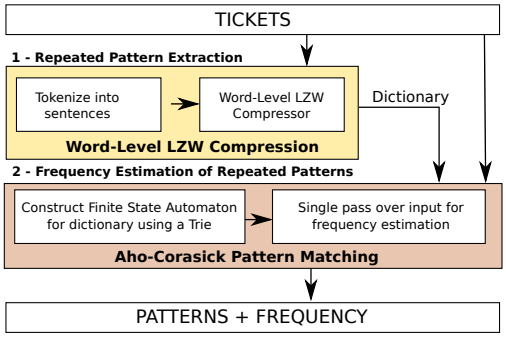


Figure 5: Two Phase Pattern Extraction. First, NetSieve tokenizes input into sentences and applies WLZW to build a dictionary of repeated patterns. Second, it uses the Aho-Corasick pattern matching algorithm to calculate their frequency.

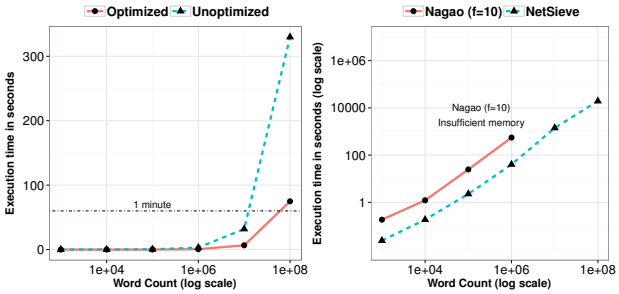


Figure 6: Performance of WLZW (a): Optimized implementation using Cython gives a performance boost of up to 5x-20x over a Python based solution as expected. Comparing NetSieve with N-gram extraction of Nagao et al. [39] (b): NetSieve is able to scale well beyond a million words in comparison to Nagao(f=10), where f is the phrase frequency.

55] is difficult when the corpus size is large.

3. Not all n-grams are *useful* due to their semantic context. For instance, n-grams such as "showing green" (LED status) and "unracking the" (unmounting the server) occurred frequently together but they do not contribute to the domain knowledge.

To address these challenges, we trade completeness in n-gram extraction for scalability and speedup. Our idea to extract hot patterns is to use a data compression algorithm, typically used to compress files by finding recurring patterns in the data and encoding them. A dictionary is maintained to map the repeated patterns to their output codes. Clearly, these dictionaries do not include all possible n-grams, but they contain hot patterns that are frequent enough to bootstrap the knowledge base.

Data compression algorithms typically operate at a byte or character level, and they do not output the frequency of patterns in their dictionary. To address these issues, NetSieve performs pattern extraction in two phases (Figure 5). First, it tokenizes input into sentences and leverages LZW [49] to develop a word-level LZW encoder (WLZW) that builds a dictionary of repeated patterns at the word-level. In the second phase, NetSieve applies the Aho-Corasick algorithm [2] to output frequency of the repeated phrases. Aho-Corasick is a string matching algorithm that runs in a single-pass and has a complexity linear in the pattern length, input size and the number of output matches.

Table 2: Examples of phrases extracted using the Two Phase Pattern Extraction algorithm.

Phrase Type	Phrase Pattern
Frequent messages	team this is to inform you that there has been a device down alarm reported on
Debug messages	errors 0 collisions 0 interface resets 0 babbles 0 late collision 0 deferred <device> sup 1a
Email snippets	if you need assistance outside of these hours please call into the htts toll free number 1 800

IMPLEMENTATION: We implemented the two phase pattern extraction algorithm in Cython [3], that allows translating Python into optimized C/C++ code. To optimize performance, we implemented the Aho-Corasick algorithm using suffix-trees [48] as opposed to the conventional suffix-arrays [30]. As expected, we achieved a 5x-20x performance improvement using Cython compared to a Python-based solution (Figure 6(a)).

Figure 6(b) shows the performance comparison of WLZW to *Nagao (f=10)* [39] which extracts all n-grams that occur at least 10 times. The latter terminated due to insufficient memory for a million word document. In comparison, NetSieve is able to process documents containing 100 million words in under 2.7 hours.

Note that WLZW is one way to extract hot patterns; we will explore other methods [16, 18, 52] in the future.

3.3.2 Knowledge Discovery

The goal of the knowledge discovery phase is to filter important domain-specific patterns from the extracted set in the previous phase; Table 2 shows examples of the extracted phrases. We define a pattern as *important* when it contributes to understanding of the "central topic" in a ticket. Consider the following excerpt from a ticket:

> We found that the **device** <**name**> **Power LED** is **amber** and it is in **hung state**. This device has **silver power supply**. We need to change the **silver power supply** to **black**. We will let you know once the **power supply** is **changed**.

The central topic of the above excerpt is "device failure that requires a power supply unit to be changed" and the

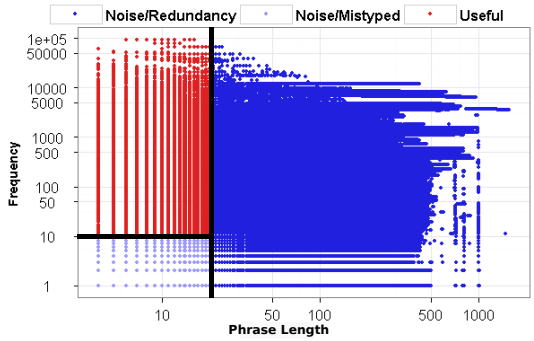


Figure 7: Filtering phrases using phrase length and frequency.

phrases in bold are relevant. While the pattern extractor outputs these phrases as repeated patterns, the key challenge is how to distinguish them from noisy patterns.

DESIGN: An intuitive method is to select the most frequently occurring patterns as important. However, we observed that many of them were warning messages which did not contribute to the central topic. Therefore, we apply a pipeline of three filters to identify the important domain-specific patterns.

Phrase Length/Frequency Filter: The idea behind applying this filter is that both the length and frequency of a phrase can act as good indicators of its importance. Intuitively, we are interested in phrases of short-length, but having a high-frequency. The rationale is that the other length-frequency combinations are either noise, occur due to typos, or can be constructed using short phrases.

We did not use a spell checker because an untrained or an undertrained one may incorrectly modify domain-specific words such as DNS and DMZ, and the probability of an important domain-specific phrase having typos in a large fraction of tickets is likely small. We plot the distribution of length and frequency of phrases (Figure 7) and then manually inspect a random subset in each quartile to derive heuristics for threshold-based filtering (Table 3).

Part-Of-Speech (PoS) Filter: The second filter is based on the seminal work of Justeson *et al.* [23]. They postulate that technical terms or domain-specific phrases have no satisfactory formal definition and can only be intuitively characterized: they generally occur only in specialized types of usage and are often specific to subsets of domains. Specifically, they conclude that most technical phrases contain only nouns and adjectives after analyzing four major technical dictionaries and subsequently provide a set of seven patterns outlined

Table 3: Thresholds for Phrase Length/Frequency filter.

Filtering Rule	Reason
Length $>$ 20 words	Likely Templates (long repeated patterns)
Single word phrases *i.e.*, unigrams	Important unigrams occur as part of a bi-gram or a tri-gram.
Frequency $<$ 10 i.e., rare words or phrases	Likely an isolated incident and not a frequently occurring problem trend
Contain numbers	Domain-specific phrases rarely contain numbers

Table 4: NetSieve converts the Justeson-Katz PoS patterns to Penn Treebank PoS patterns to filter technical phrases.

Justeson-Katz Patterns	NetSieve Patterns	Example
Adjective Noun	JJ NN[PS]*	mobile network
Noun Noun	NN[PS]* NN[PS]*	demo phase
Adjective Adjective Noun	JJ JJ NN[PS]*	fast mobile network
Adjective Noun Noun	JJ NN[PS]* NN[PS]*	accessible device logs
Noun Adjective Noun	NN[PS]* JJ NN[PS]*	browser based authentication
Noun Noun Noun	NN[PS]* NN[PS]* NN[PS]*	power supply unit
Noun Preposition Noun	NN[PS]* IN NN[PS]*	device down alert
JJ: Adjective; **NN**: Singular Noun; **NNP**: Proper singular noun; **NNPS**: Proper plural noun; **IN**: Preposition		

in Table 4. We build upon these patterns and map them to state-of-the-art Penn Treebank tagset [34], a simplified part-of-speech tagset for English, using regular expressions (Table 4). Further, this mapping allows our implementation to leverage existing part-of-speech taggers of natural language toolkits such as NLTK [29] and SharpNLP [44]. Filtering takes place in two steps: (1) each input phrase is tagged with its associated part-of-speech tags, and (2) the part-of-speech pattern is discarded if it fails to match a pattern.

Entropy Filter: The third filter uses information theory to filter statistically insignificant phrases, and sorts them based on importance to aid manual labeling. We achieve this by computing two metrics for each phrase [52]:

1. **Mutual Information (MI)**: $MI(x,y)$ compares the probability of observing word x and word y together (the joint probability) with the probabilities of observing x and y independently. For a phrase pattern, MI is computed by the following formula:

$$MI(xYz) = \log\left(\frac{tf(xYz) * tf(Y)}{tf(xY) * tf(Yz)}\right) \quad (1)$$

where xYz is a phrase pattern, x and z are a word/char-

acter and Y is a sub-phrase or sub-string, tf denotes the *term-frequency* of a word or phrase in the corpus.

2. **Residual Inverse Document Frequency (RIDF)**: RIDF is the difference between the observed IDF and what would be expected under a Poisson model for a random word or phrase with comparable frequency. RIDF of a phrase is computed as follows:

$$RIDF = -\log\left(\frac{df}{D}\right) + \log\left(1 - \exp\left(\frac{-tf}{D}\right)\right) \quad (2)$$

where df denotes the document-frequency (the number of tickets which contain the phrase) and D as the total number of tickets.

Phrases with high RIDF or MI have distributions that cannot be attributed to chance [52]. In particular, MI aims to pick vocabulary expected in a dictionary, while RIDF aims to select domain-specific keywords, not likely to exist in a general dictionary. We investigated both metrics as they are orthogonal and that each tends to separately pick interesting phrases [52].

IMPLEMENTATION: The three filters are applied as a sequential pipeline and are implemented in Python. For PoS tagging, we utilize the Stanford Log-Linear PoS Tagger [46] as an add-on module to the Natural Language Toolkit [29] and implement a multi-threaded tagger that uses the phrase length/frequency filter to first filter a list of candidate phrases for tagging.

After applying the threshold-based filtering and PoS filters on the input 18.85M phrases, RIDF and MI are computed for the remaining 187K phrases. This step significantly reduced the computational cost compared to prior work [9, 39, 45, 52], which aim to compute statistics for all n-grams. Similar to [52], we did not observe strong correlation between RIDF and MI (Figure 8 (top)). We relied solely on RIDF because most phrases with high MI were already filtered by RIDF and the remaining ones contained terms not useful in our context.

The bottom graph of Figure 8 shows the CCDF plot of RIDF which can be used to set a threshold to narrow down the phrase list for the next stage of human review. Determining the threshold poses a trade off between the completeness of the domain-dictionary and the human effort required to analyze the extracted patterns. In our prototype, we set the threshold based on RIDF such that 3% (5.6K) of the total phrases (187K) are preserved. Further, we sort these phrase patterns based on RIDF for expert review as phrases with higher values get labeled quickly. An expert sifted through the 5.6K phrases (Figure 9) and selected 1.6K phrase patterns that we consider as ground truth, in less than four hours. We observed that

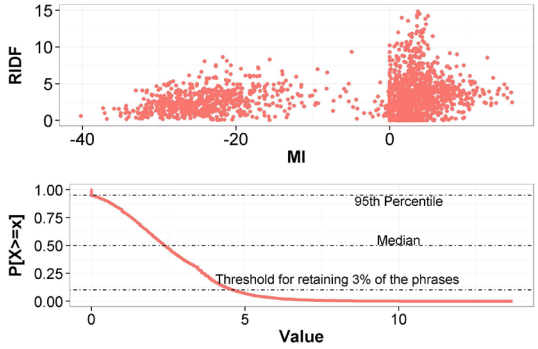


Figure 8: Absence of correlation between RIDF and MI metrics (top). Using a CCDF plot of RIDF to determine a threshold for filtering the phrases for expert review (bottom).

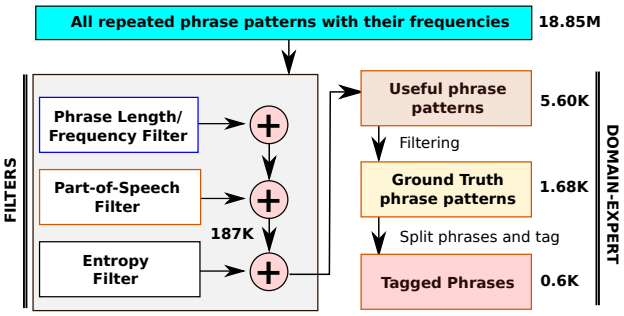


Figure 9: The pipeline of filtering phrases to determine a list of ground truth phrase patterns which are then split and tagged manually with the NetSieve-Ontology classes.

most of the discarded patterns were redundant as they can be constructed from the ground truth patterns.

While we leverage manual review to obtain the ground truth, this is a necessary step for any supervised technique. We plan to explore other techniques such as Named-Entity Recognition [35] and using domain experts for crowdsourcing [26] to automate this step.

3.3.3 Ontology Modeling

The goal of building ontology models is to determine *semantic interpretation* of important domain-specific phrases generated by the knowledge discovery stage. For instance, between the terms *slot* and *memory slot*, we are looking for the latter term with high specificity. Intuitively, we need to model an ontology where domain-specific phrases have a concrete meaning and can be combined together to enable semantic interpretation.

DESIGN: Developing an ontology model involves three steps [17, 40]: (1) defining classes in the ontology, (2) arranging the classes in a taxonomic (superclass, subclass) hierarchy and (3) defining interactions amongst the

Table 5: Classes in the NetSieve-Ontology model

Class	Sub-Class	Interpretation	Example
Entity	Replaceable	Tangible object that can be created/destroyed/replaced	Flash memory, Core router
	Virtual	Intangible object that can be created/destroyed/replaced	Port channel, configuration
	Maintenance	Tangible object that can act upon other entities	field engineer, technician
Action	Physical	Requires creating/destroying an entity	decommission, replace, rma
	Maintenance	Requires interacting with an entity and altering its state	clean, deploy, validate, verify
Condition	Problem	Describes condition that is known to have a negative effect	inoperative, reboot loop
	Maintenance	Describes condition that describes maintenance	break-fix
Incident	False Positive	State known to not have any problems	false positive, false alert
	Error	State known to cause a problem	error, exception
Quantity	Low	Describes the quantity of an entity/action	low, minor
	Medium		medium
	High		high, major
Negation	Synthetic	Uses verb or noun to negate a condition/incident/action	absence of, declined, denied
	Analytic	Uses 'not' to negate a condition/incident/action	not
Sentiment	Positive	Adds strength/weakness to an action/incident	confirm, affirmative
	Neutral		not sure
	Negative		likely, potential

classes. Note that defining an ontology is highly domain-specific and depends on extensions anticipated by the domain-expert. We designed and embedded one such ontology into NetSieve based on discussion with operators. Below, we discuss the design rationale behind the classes, taxonomies and interactions in our model.

Classes and Taxonomies: A class describes a concept in a given domain. For example, a class of entities can represent all devices (*e.g.*, routers, load balancers and cables) and a class of conditions can represent all possible states of an entity (*e.g.*, bad, flapping, faulty). A domain-expert sifted through the 1.6K phrases from previous stage and after a few iterations, identified seven classes to describe the phrases, shown in Table 5.

Taxonomic Hierarchy: To enable fine-grained problem inference, *Entity* is divided into three sub-classes: *Replaceable* denoting entities that can be physically replaced, *Virtual* denoting entities that are intangible and *Maintenance* denoting entities that can "act" upon other entities. *Actions* are sub-classed in a similar way. The rest of the classes can be considered as qualifiers for *Entities* and *Actions*. Qualifiers, in general, act as adjectives or adverbs and give useful information about an *Entity* or *Action*. In the final iteration, our domain-expert split each of the 1.6K long phrase patterns into their constituent small phrase patterns and tagged them with the most specific class that captured the phrase e.g.,
"and gbic replacement" → [(and, OMIT WORD), (gbic, ReplaceableEntity),(replacement, PhysicalAction)].

Most of these tagged phrases are domain-specific multi-word phrases and are not found in a dictionary. While the words describing Entities were not ambiguous, we found a few cases where other classes were ambiguous. For instance, phrases such as "power supply" (hardware unit or power line), "bit errors" (memory or network link), "port channel" (logical link bundling or virtual link), "flash memory" (memory reset or type of memory), "device reload" and "interface reset" (unexpected or planned), and "key corruption" (crypto-key or license-key) were hard to understand without a proper context. To address this ambiguity, we use the text surrounding these phrases to infer their intent (§3.4).

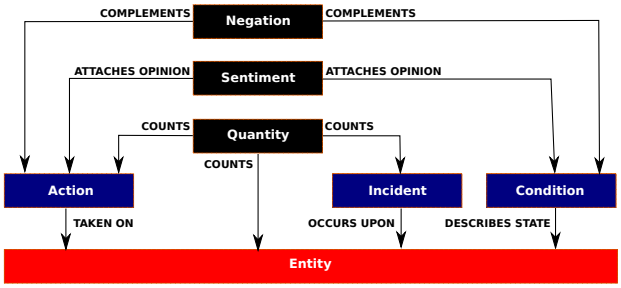


Figure 10: Ontology Model depicting interactions amongst the NetSieve-Ontology Classes.

Finally, using these mappings, NetSieve embeds a *ClassTagger* module that given an input, outputs tags for words that have an associated class mapping.

Interactions: An interaction describes relationships amongst the various classes in the ontology model. For instance, there are valid interactions (an *Action* can be caused upon an *Entity*) and invalid interactions (an *Entity* cannot describe a *Sentiment*). Figure 10 shows our model comprising interactions amongst the classes.

IMPLEMENTATION: We obtained 0.6K phrases from the 1.6K phrases in §3.3.2 categorized into the seven classes. We implemented the *ClassTagger* using a trie constructed using NetSieve knowledge base of domain-

Table 6: Concepts for the NetSieve-Ontology

Concept	Pattern	Example
Problems	[Replaceable — Virtual — Maintenance] Entity preceded/succeeded by ProblemCondition	The (device) was (faulty)
Activities	[Replaceable — Virtual — Maintenance] Entity preceded/succeeded by MaintenanceAction	(check) (device) connectivity and (clean) the (fiber)
Actions	[Replaceable — Virtual — Maintenance] Entity preceded/succeeded by PhysicalAction	An (RMA) was initiated for the (load balancer)

specific phrases, and a dictionary of their ontology mappings. The tagging procedure works in three steps. First, the input is tokenized into sentences. Second, using the trie, a search is performed for the longest matching phrase in each sentence to build a list of domain-specific keywords e.g., in the sentence "the power supply is down", both "supply" and "power supply" are valid domain keywords, but the ClassTagger marks "power supply" as the relevant word. Finally, these keywords are mapped to their respective ontology classes using the dictionary. For instance, given the snippet from §3.3.2, the *ClassTagger* will produce the following output:

We found that the (device) **/ ReplaceableEntity** <**name**> **(Power LED) / ReplaceableEntity** is **(amber) / Condition** and it is in **(hung state) / ProblemCondition**. This device has **(silver) / Condition (power supply) / ReplaceableEntity**. We need to change the **(silver) / Condition (power supply) / ReplaceableEntity** to **(black) / Condition**. We will let you know once the **(power supply) / ReplaceableEntity** is **(changed) / PhysicalAction**.

3.4 Operational Phase

The goal of this phase is to leverage the knowledge base to do automated problem inference on trouble tickets. A key challenge to address is how to establish a relationship between the ontology model and the physical world. In particular, we want to map certain interactions from our ontology model to concepts that allow summarizing a given ticket.

DESIGN: Our discussion with operators revealed a common ask to answer three main questions: (1) What was observed when a problem was logged?, (2) What activities were performed as part of troubleshooting? and (3) What was the final action taken to resolve the problem? Based on these requirements, we define three key concepts that can be extracted using our ontology model (Table 6): (1) *Problems* denote the state or condition of an entity, (2) *Activities* describe the troubleshooting steps, and (3) *Actions* capture the problem resolution.

The structure of concepts can be identified by sampling tickets describing different types of problems. We randomly sampled 500 tickets out of our expert-labeled ground truth data describing problems related to different device and link types. We pass these tickets through NetSieve's *ClassTagger* and get a total of 9.5K tagged snippets. We observed a common linguistic structure in them: in more than 90% of the cases, the action/condition that relates to an entity appears in the same sentence *i.e.,* information can be inferred about an entity based on its neighboring words. Based on this observation, we derived three patterns (Table 6) that capture all the cases of interest. Intuitively, we are interested in finding instances where an action or a condition precedes/succeeds an entity. Based on the fine granularity of the sub-classes, the utility of the concepts extracted increases i.e., *PhysicalAction was taken on a ReplacableEntity* is more important than *Action was taken on an Entity*.

This type of a proximity-search is performed once for each of the three concepts. First, the *ClassTagger* produces a list of phrases along with their associated tags. Second, we check to see if the list of tags contain an action/condition. Once such a match is found in a sentence, the phrase associated with the action/condition is added to a dictionary as a key and all entities within its neighborhood are added as corresponding values. We implemented several additional features like negation detection [15] and synonym substitution to remove any ambiguities in the inference output.

EXAMPLE: *"The load balancer was down. We checked the cables. This was due to a faulty power supply unit which was later replaced"*, is tagged as *"The (load balancer) / ReplaceableEntity was (down) / ProblemCondition. We (checked) / MaintenanceAction the (cables) / ReplaceableEntity. This was due to a (faulty) / ProblemCondition (power supply unit) / ReplaceableEntity which was later (replaced) / PhysicalAction"*. Next, a dictionary is built for each of the three concepts. Two classes are associated if they are direct neighbors. In this example, the output is the following:
[+] Problems - {down: load balancer, power supply unit}
[+] Activities - {checked: cable}
[+] Actions - {replaced: power supply unit}
In the final stage, the word "replaced" is changed into "replace" and "checked" into "check" using a dictionary of synonyms provided by the domain-expert to remove any ambiguities.

IMPLEMENTATION: We implemented our inference logic using Python. Each ticket is first tokenized into sentences and each sentence is then used for concept inference. After extracting the concepts and their associated entities, we store the results in a SQL table. Our implementation is able to do problem inference at the rate of 8 tickets/sec on average on our server, which scales to 28,800 tickets/hour; note that this time depends on the ticket size, and the overhead is mainly due to text processing and part-of-speech tagging.

Table 7: Evaluating NetSieve accuracy using different datasets. High F-scores are favorable.

Source	Dataset		Precision		Recall		F-Score		Accuracy %	
	Devices	# Tickets	Problems	Actions	Problems	Actions	Problems	Actions	Problems	Actions
Domain Expert	LB-1	122	1	1	0.982	0.966	0.991	0.982	97.7	96.6
	LB-2	62	1	1	1	1	1	1	100.0	100.0
	LB-3	31	1	1	1	0.958	1	0.978	100.0	95.7
	LB-4	36	1	1	1	1	1	1	100.0	100.0
	FW	35	1	1	0.971	0.942	0.985	0.970	97.1	94.3
	AR	410	1	1	0.964	0.951	0.981	0.974	96.4	95.1
Vendor	LB	78	1	1	0.973	0.986	0.986	0.993	97.3	98.7
	CR	77	1	1	1	0.896	1	0.945	100.0	89.6
CR: Core Routers; **LB[1-4]**:Types of Load balancers; **FW**:Firewalls; **AR**: Access Routers										

4 Experimental Results

For evaluation, we use two standard metrics from information retrieval: (1) *Accuracy Percentage* [31] computed as $\frac{TP+TN}{TP+TN+FP+FN}$ and (2) *F-Score* [32] computed as $\frac{2TP}{2TP+FP+FN}$, where TP, TN, FP, FN are true positives, true negatives, false positives and false negatives, respectively. F-Scores consider both precision and recall, and its value of 1 indicates a perfect classifier. Precision is defined as the ratio of TP and (TP+FP), and recall is defined as the ratio of TP and (TP+FN).

4.1 Evaluating NetSieve Accuracy

To test NetSieve's accuracy, we randomly divided our two year ticket dataset into training and test data. The test data consists of 155 tickets on device replacements and repairs from two network vendors and 696 tickets labeled by a domain expert while the training data comprises the rest of the tickets. We use the training data to build NetSieve's knowledge base. The domain expert read the original ticket to extract the ground truth in terms of Problems and Actions, which was then compared with corresponding phrases in the inference output.

Table 7 shows the overall results by NetSieve on the test dataset. On the expert labeled data, we observe the precision of NetSieve to be 1, minimum recall value to be 0.964 for Problems and 0.942 for Actions, F-score of 0.981-1 for Problems and 0.970-1 for Actions, and accuracy percentage to be 96.4%-100% for Problems and 94.3%-100% for Actions. These results indicate that NetSieve provides useful inference with reasonable accuracy in analyzing network tickets.

Next, we validate the inference output against data from two network device vendors that record the ground truth based on the diagnosis of faulty devices or components sent back to the vendor. Each vendor-summary reported the root cause of the failure (similar to NetSieve's Problems) and what was done to fix the problem (similar to NetSieve's Actions). We obtained vendor data corresponding to total 155 tickets on load balancers and core routers from our dataset. Since the vocabulary in the vendor data comprised new, vendor-specific words and synonyms not present in our knowledge base (e.g., port interface mentioned as 'PME'), we asked a domain-expert to validate if NetSieve's inference summary covers the root cause and the resolution described in the vendor data. For instance, if the vendor data denoted that a router failed due to a faulty supervisor engine (SUP), the expert checked if NetSieve captures "failed device" under *Problems* and "replaced SUP" under *Actions*.

The accuracy of NetSieve on the vendor data is observed to be 97.3%-100% for Problems and 89.6%-100% for Actions. One reason for relatively lower accuracy for Actions on this dataset is due to a small number of false negatives: the corrective action applied at the vendor site may differ from our ticket set as the vendor has the expert knowledge to fix problems specific to their devices.

Overall, NetSieve has reasonable accuracy of 96.4%-100% for *Problems* and 89.6%-100% for *Actions*, measured based on the labeled test dataset. We observed similar results for *Activities* and thus omit them.

4.2 Evaluating Usability of NetSieve

We conducted a user study involving five operators to evaluate the usability of NetSieve for automated problem inference compared to the traditional method of manually analyzing tickets. Each operator was shown 20 tickets selected at random from our dataset and asked to analyze the type of problems observed, activities and actions. Then, the operator was shown NetSieve inference summary of each ticket and asked to validate it against their manually labeled ground truth. We measured the accuracy, speed and user preference in the survey.

Figure 11 shows the accuracy and time to manually read the ticket versus the NetSieve inference summary across the operators. We observed average accuracy of 83%-100% and a significant decrease in time taken to manually read the ticket from 95P of 480s to 95P of 22s for NetSieve.

Table 8: Top-3 Problems/Activities/Actions and Failing Components as obtained through NetSieve's Trend Analysis

Device	Problems	Activities	Actions	Failing Components
AR	memory error, packet errors, illegal frames	verify cable, reseat/clean cable, upgrade OS	replace with spare, rma, reboot	SUP engine, cables, memory modules
AGG	device failure, packet errors, defective N/W card	verify cable, upgrade N/W card, swap cable	replace with spare, reboot, rma	cables, N/W card, SUP engine
CR	circuit failure, N/W card failure, packet errors	verify cable, verify N/W card, upgrade fiber	replace with spare, reboot, rma	cables, N/W, memory modules
ER	circuit failure, N/W card failure, packet errors	verify cable, verify N/W card, upgrade fiber	replace with spare, rma, reboot	N/W card, chassis, cables
LB	PSU failure, device rebooted, config error	verify PSU, verify config, verify cable	replace with spare, reboot, rma	PSU, HDD, memory modules
ToR	connection failure, ARP conflict, SUP engine failure	verify cable, power cycle blade, verify PSU	reboot, replace with spare, rma	cables, OS, SUP engine
FW	connection failure, reboot loop, data errors	verify config, verify connections, verify PSU	reboot, replace with spare, rma	cables, PSU, N/W card
VPN	connection failure, config error, defective memory	verify config, verify N/W card, deploy N/W card	reboot, replace with spare, rma	OS, SUP engine, memory modules

AR: Access Routers, **AGG**: Aggregation Switch; **[E/C]R**: Edge/Core Router; **LB**: Load Balancer; **ToR**: Top-of-Rack Switch; **FW**: Firewall

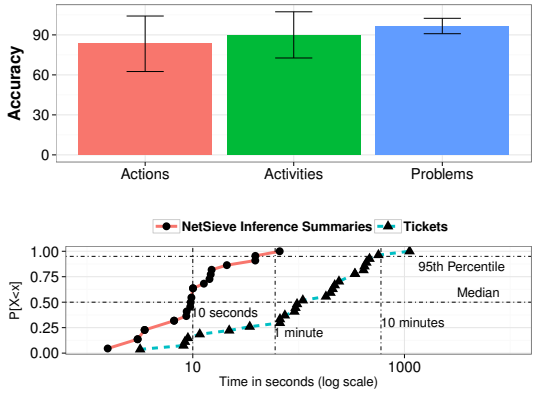


Figure 11: Accuracy obtained from the user survey (top). CDF of time to read tickets and inference summaries (bottom).

5 Deployment and Impact

NetSieve has been deployed in the cloud provider we studied to enable operators to understand global problem trends instead of making decisions based on isolated incidents. Further, NetSieve complements existing tools (e.g., inventory db) by correlating device replacements with their failures and problem root causes. A simple alternative to mining tickets using NetSieve is to ask operators for explicit feedback e.g., to build Table 8, but it will likely be biased by anecdotal or recent data.

Currently, our NetSieve prototype supports SQL-like queries on the inference output. For instance, "SELECT TOP 5 Problems FROM InferenceDB WHERE DeviceType = 'Load Balancer' would output the top-5 failure problems observed across load balancers. Next, we present how NetSieve has been used across different teams to improve network management.

Network Architecture: This team used NetSieve to compare device reliability across platforms and vendors. In one instance, NetSieve showed that a new generation of feature-rich, high capacity AR is half as reliable as its predecessor. In another instance, it showed that software bugs dominated failures in one type of load balancers. Based on grouping tickets having Problem inference of Switch Card Control Processor (SCCP) watchdog timeout for LB-2, NetSieve showed that hundreds of devices exhibited reboot loops due to a recurring software bug and were RMAed in 88% of the tickets.

Capacity Planning: This team applied NetSieve to analyze cases when network redundancy is ineffective in masking failures. Specifically, for each network failure, we evaluate if the redundancy failover was not successful [13] and then select the corresponding tickets. These tickets are then input to NetSieve to do problem inference which output the following to be the dominant problems:

1. **Faulty Cables**: The main reason was that the cable connected to the backup was faulty. Thus, when the primary failed, it resulted in a high packet error rate.
2. **Software Mismatch**: When the primary and backup had mismatched OS versions, protocol incompatibility caused an unsuccessful failover.
3. **Misconfiguration**: Because operators usually configure one device and then copy the configuration onto the other, a typo in one script introduced the same bug in the other and resulted in an unsuccessful failover.
4. **Faulty Failovers**: The primary device failed when the backup was facing an unrelated problem such as software upgrade, down for scheduled repairs, or while deploying a new device into production.

Incident Management and Operations: The incident management team used NetSieve to prioritize checking for the top-k problems and failing components while

troubleshooting devices. The operation team uses NetSieve to determine if past repairs were effective and decide whether to repair or replace the device. Table 8 shows NetSieve's inference output across different device types.We observe that while Problems show a high diversity across device types such as packet errors, line card failures and defective memory, verifying cable connectivity is a common troubleshooting activity, except for Firewalls and VPNs where operators first verify device configuration. For Actions related to Firewalls, VPNs and ToRs, the devices are first rebooted as a *quick fix* even though the failing components are likely to be bad cable or OS bugs. In many cases, we observe that a failed device is RMAed (sent back to the vendor) which implies that the network is operating at reduced or no redundancy until the replacement arrives.

Complementary to the above scenarios, NetSieve inference can be applied in several other ways: (1) prioritize troubleshooting steps based on frequently observed problems on a given device, its platform, its datacenter, or the hosted application, (b) identify the top-k failing components in a device platform and resolving them with their vendors, and (c) decide whether to repair, replace or even retire a particular device or platform by computing a total cost-of-ownership (TCO) metric.

6 Related Work

Network Troubleshooting: There has been a significant work in analyzing structured logs to learn statistical signatures [1, 10], run-time states [54] or leveraging router syslogs to infer problems [41]. Xu et al. [51] mine machine logs to detect runtime problems by leveraging the source code that generated the logs. NetSieve complements these approaches to automate problem inference from unstructured text. Expert systems [8, 25], on the other hand, diagnose network faults based on a set of pre-programmed rules. However, they lack generality as they only diagnose faults in their ruleset and the ruleset may become outdated as the system evolves. In comparison, the incremental learning phase in NetSieve updates the knowledge base to improve the inference accuracy.

Mining Network Logs: Prior efforts have focused on automating mining of network failures from syslogs [41, 53] or network logs [28]. However, these studies do not analyze free-form text in trouble tickets. Kandula et al. [24] mine rules in edge networks based on traffic data. Brauckhoff et al. [7] use association rule mining techniques to extract anomalies in backbone networks. NetSieve is complementary to these efforts in that their mining methodologies can benefit from our domain-specific knowledge base. TroubleMiner [36] selects keywords manually from the first two sentences in tickets and then performs clustering to group them.

Analyzing Bug Reports: There is a large body of work in software engineering analyzing [19, 22], summarizing [6] and clustering [5, 43] bug reports. Betternburg et al. [6] rely on features found in bug reports such as stack traces, source code, patches and enumerations and, hence their approach is not directly applicable to network tickets. Others [5, 43] use standard NLP techniques for the task of clustering duplicate bug reports, but they suffer from the same limitations as keyword based approaches. In comparison, NetSieve aims to infer "meaning" from the free-form content by building a knowledge base and an ontology model to do problem inference.

Natural Language Processing: *N*-gram extraction techniques [9, 39, 45, 52] focus on extracting all possible *n*-grams thereby incurring a high computation cost for large datasets (§3.3.1). NetSieve addresses this challenge by trading completeness for scalability and uses the dictionary built by its WLZW algorithm. Wu et al. [50] detect frequently occurring text fragments that have a high correlation with labels in large text corpora to detect issues in customer feedback. Other efforts [14, 38] achieve summarization via paragraph and sentence extraction. Much research in this area deals with properly-written regular text and is not directly applicable to our domain. In contrast, NetSieve focuses on free-form text in trouble tickets to do problem inference.

7 Conclusion

Network trouble tickets contain valuable information for network management, yet they are extremely difficult to analyze due to their free-form text. This paper takes a practical approach towards automatically analyzing the natural language text to do problem inference. We presented NetSieve that automatically analyzes ticket text to infer the problems observed, troubleshooting steps, and the resolution actions. Our results are encouraging: NetSieve achieves reasonable accuracy, is considered useful by operators and has been applied to answer several key questions for network management.

8 Acknowledgements

We thank our shepherd Kobus Van der Merwe and the anonymous reviewers for their feedback; our colleagues Sumit Basu, Arnd Christian König, Vivek Narasayya, Chris Quirk and Alice Zheng for their insightful comments; and the network operations team for their guidance and participation in our survey.

References

[1] AGUILERA, M., MOGUL, J., WIENER, J., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *ACM SIGOPS Operating Systems Review* (2003), ACM.

[2] AHO, A., AND CORASICK, M. Efficient string matching: an aid to bibliographic search. *Communications of the ACM* (1975).

[3] BEHNEL, S., BRADSHAW, R., SELJEBOTN, D., EWING, G., ET AL. Cython: C-extensions for python, 2008.

[4] BENSON, M. Collocations and general-purpose dictionaries. *International Journal of Lexicography* (1990).

[5] BETTENBURG, N., PREMRAJ, R., ZIMMERMANN, T., AND KIM, S. Duplicate bug reports considered harmful really? In *IEEE International Conference on Software Maintenance* (2008).

[6] BETTENBURG, N., PREMRAJ, R., ZIMMERMANN, T., AND KIM, S. Extracting structural information from bug reports. In *ACM International Working Conference on Mining Software Repositories* (2008).

[7] BRAUCKHOFF, D., DIMITROPOULOS, X., WAGNER, A., AND SALAMATIAN, K. Anomaly extraction in backbone networks using association rules. In *ACM Internet Measurement Conference* (2009).

[8] BRUGNONI, S., BRUNO, G., MANIONE, R., MONTARIOLO, E., PASCHETTA, E., AND SISTO, L. An expert system for real time fault diagnosis of the italian telecommunications network. In *International Symposium on Integrated Network Management* (1993).

[9] CHURCH, K., AND HANKS, P. Word association norms, mutual information, and lexicography. *Journal of Computational Linguistics* (1990).

[10] COHEN, I., ZHANG, S., GOLDSZMIDT, M., SYMONS, J., KELLY, T., AND FOX, A. Capturing, indexing, clustering, and retrieving system history. In *ACM SIGOPS Operating Systems Review* (2005), ACM.

[11] FIELDING, R. Representational state transfer (rest). *Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine* (2000), 120.

[12] FORD, D., LABELLE, F., POPOVICI, F., STOKELY, M., TRUONG, V., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in globally distributed storage systems. In *USENIX Symposium on Operating Systems Design and Implementation* (2010).

[13] GILL, P., JAIN, N., AND NAGAPPAN, N. Understanding network failures in data centers: measurement, analysis, and implications. In *SIGCOMM Computer Communication Review* (2011).

[14] GOLDSTEIN, J., KANTROWITZ, M., MITTAL, V., AND CARBONELL, J. Summarizing text documents: sentence selection and evaluation metrics. In *International ACM SIGIR Conference on Research and Development in Information Retrieval* (1999).

[15] GORYACHEV, S., SORDO, M., ZENG, Q., AND NGO, L. Implementation and evaluation of four different methods of negation detection. Tech. rep., DSG, 2006.

[16] GOYAL, A., DAUMÉ III, H., AND VENKATASUBRAMANIAN, S. Streaming for large scale NLP: Language modeling. In *Annual Conference of the Association for Computational Linguistics* (2009).

[17] GRUBER, T., ET AL. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human Computer Studies* (1995).

[18] HEAFIELD, K. Kenlm: Faster and smaller language model queries. In *Workshop on Statistical Machine Translation* (2011).

[19] HOOIMEIJER, P., AND WEIMER, W. Modeling bug report quality. In *IEEE/ACM International Conference on Automated Software Engineering* (2007).

[20] HUANG, Y., FEAMSTER, N., LAKHINA, A., AND XU, J. Diagnosing network disruptions with network-wide analysis. In *ACM SIGMETRICS Performance Evaluation Review* (2007).

[21] JOHNSON, D. Noc internal integrated trouble ticket system. `http://goo.gl/eMZxX`, January 1992.

[22] JUST, S., PREMRAJ, R., AND ZIMMERMANN, T. Towards the next generation of bug tracking systems. In *IEEE Symposium on Visual Languages and Human-Centric Computing* (2008).

[23] JUSTESON, J., AND KATZ, S. Technical terminology: some linguistic properties and an algorithm for identification in text. *Journal of Natural language engineering* (1995).

[24] KANDULA, S., CHANDRA, R., AND KATABI, D. What's going on?: learning communication rules in edge networks. *ACM SIGCOMM Computer Communication Review* (2008).

[25] KHANNA, G., CHENG, M., VARADHARAJAN, P., BAGCHI, S., CORREIA, M., AND VERÍSSIMO, P. Automated rule-based diagnosis through a distributed monitor system. *IEEE Transactions on Dependable and Secure Computing* (2007).

[26] KITTUR, A., CHI, E., AND SUH, B. Crowdsourcing user studies with mechanical turk. In *ACM SIGCHI Conference on Human factors in Computing Systems* (2008).

[27] LABOVITZ, C., AHUJA, A., AND JAHANIAN, F. Experimental study of internet stability and backbone failures. In *IEEE International Symposium on Fault-Tolerant Computing* (1999).

[28] LIM, C., SINGH, N., AND YAJNIK, S. A log mining approach to failure analysis of enterprise telephony systems. In *IEEE Dependable Systems and Networks* (2008).

[29] LOPER, E., AND BIRD, S. Nltk: The natural language toolkit. In *Association for Computational Linguistics Workshop on Effective Tools and Methodologies for teaching Natural Language Processing and Computational Linguistics* (2002).

[30] MANBER, U., AND MYERS, G. Suffix arrays: a new method for on-line string searches. In *ACM SIAM Journal on Computing* (1990).

[31] MANI, I., HOUSE, D., KLEIN, G., HIRSCHMAN, L., FIRMIN, T., AND SUNDHEIM, B. The tipster summac text summarization evaluation. In *Association for Computational Linguistics* (1999).

[32] MANNING, C., RAGHAVAN, P., AND SCHUTZE, H. *Introduction to information retrieval*, vol. 1. Cambridge University Press Cambridge, 2008.

[33] MANNING, C., AND SCHÜTZE, H. *Foundations of statistical natural language processing*. MIT Press, 1999.

[34] MARCUS, M., MARCINKIEWICZ, M., AND SANTORINI, B. Building a large annotated corpus of english: The penn treebank. *MIT Press Computational Linguistics* (1993).

[35] MCCALLUM, A., AND LI, W. Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons. In *Association for Computational Linguistics Conference on Natural Language Learning* (2003).

[36] MEDEM, A., AKODJENOU, M., AND TEIXEIRA, R. Troubleminer: Mining network trouble tickets. In *IEEE International Workshop Symposium on Integrated Network Management* (2009).

[37] MELCHIORS, C., AND TAROUCO, L. Troubleshooting network faults using past experience. In *IEEE Network Operations and Management Symposium* (2000).

[38] MITRAY, M., SINGHALZ, A., AND BUCKLEYYY, C. Automatic text summarization by paragraph extraction. *Compare* (1997).

[39] NAGAO, M., AND MORI, S. A new method of n-gram statistics for large number of n and automatic extraction of words and phrases from large text data of japanese. In *ACL Computational Linguistics* (1994).

[40] NOY, N., MCGUINNESS, D., ET AL. Ontology development 101: A guide to creating your first ontology. Tech. rep., Stanford knowledge systems laboratory technical report KSL-01-05 and Stanford medical informatics technical report SMI-2001-0880, 2001.

[41] QIU, T., GE, Z., PEI, D., WANG, J., AND XU, J. What happened in my network: mining network events from router syslogs. In *ACM Internet Measurement Conference* (2010).

[42] ROUGHAN, M., GRIFFIN, T., MAO, Z., GREENBERG, A., AND FREEMAN, B. Ip forwarding anomalies and improving their detection using multiple data sources. In *ACM SIGCOMM Workshop on Network Troubleshooting: Research, Theory and Operations Practice meet malfunctioning reality* (2004).

[43] RUNESON, P., ALEXANDERSSON, M., AND NYHOLM, O. Detection of duplicate defect reports using natural language processing. In *IEEE International Conference on Software Engineering* (2007).

[44] S., D. Sharp NLP. `http://goo.gl/Im4BK`, December 2006.

[45] SMADJA, F. Retrieving collocations from text: Xtract. *MIT Press Computational linguistics* (1993).

[46] TOUTANOVA, K., AND MANNING, C. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *ACL SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora* (2000).

[47] TURNER, D., LEVCHENKO, K., SNOEREN, A., AND SAVAGE, S. California fault lines: understanding the causes and impact of network failures. In *ACM SIGCOMM Computer Communication Review* (2010).

[48] UKKONEN, E. On-line construction of suffix trees. *Algorithmica* (1995).

[49] WELCH, T. Technique for high-performance data compression. *Computer 17*, 6 (1984), 8–19.

[50] WU, F., AND WELD, D. Open information extraction using wikipedia. In *Computational Linguistics* (2010).

[51] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. Detecting large-scale system problems by mining console logs. In *ACM SIGOPS Symposium on Operating Systems Principles* (2009).

[52] YAMAMOTO, M., AND CHURCH, K. Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. *Computational Linguistics* (2001).

[53] YAMANISHI, K., AND MARUYAMA, Y. Dynamic syslog mining for network failure monitoring. In *ACM SIGKDD International Conference on Knowledge Discovery in Data Mining* (2005).

[54] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. Sherlog: error diagnosis by connecting clues from run-time logs. In *ACM SIGARCH Computer Architecture News* (2010).

[55] ZHANG, J., GAO, J., AND ZHOU, M. Extraction of chinese compound words: An experimental study on a very large corpus. In *ACL Workshop on Chinese Language Processing* (2000).

[56] ZHANG, L. N-Gram Extraction Tools. `http://goo.gl/7gGF1`, April 2012.

Yank: Enabling Green Data Centers to Pull the Plug

Rahul Singh, David Irwin, Prashant Shenoy, and K.K. Ramakrishnan[‡]
University of Massachusetts Amherst [‡]AT&T Labs - Research

Abstract

Balancing a data center's reliability, cost, and carbon emissions is challenging. For instance, data centers designed for high availability require a continuous flow of power to keep servers powered on, and must limit their use of clean, but intermittent, renewable energy sources. In this paper, we present Yank, which uses a transient server abstraction to maintain server availability, while allowing data centers to "pull the plug" if power becomes unavailable. A transient server's defining characteristic is that it may terminate anytime after a brief advance warning period. Yank exploits the advance warning—on the order of a few seconds—to provide high availability cheaply and efficiently at large scales by enabling each backup server to maintain "live" memory and disk snapshots for many transient VMs. We implement Yank inside of Xen. Our experiments show that a backup server can concurrently support up to 15 transient VMs with minimal performance degradation with advance warnings as small as 10 seconds, even when VMs run memory-intensive interactive web applications.

1 Introduction

Despite continuing improvements in energy efficiency, data centers' demand for power continues to rise, increasing by an estimated 56% from 2005-2010 and accounting for 1.7-2.2% of electricity usage in the United States [16]. The rise in power usage has led data centers to experiment with the design of their power delivery infrastructure, including the use of renewable energy sources [3, 20, 21]. For instance, Apple's goal is to run its data centers off 100% renewable power; its newest data center includes a 40MW co-located solar farm [3].

Thus, determining the characteristics of the power infrastructure—its reliability, cost, and carbon emissions—has now become a key element of data center design [5]. Balancing these characteristics is challenging, since providing a reliable supply of power is often antithetical to minimizing costs (capital or operational) and emissions. A state-of-the-art power delivery infrastructure designed to ensure an uninterrupted flow of high quality power is expensive, possibly including i) connections to multiple power grids, ii) on-site backup generators, and iii) an array of universal power supplies (UPSs) that both condition grid power and guarantee enough time after an outage to spin-up and transition power to generators. Unfortunately, while renewable energy has no emissions, it is unreliable—generating power only intermittently based on uncontrollable environmental conditions—which limits its broad adoption in data centers designed for high reliability.

Prior research focuses on balancing a data center's reliability, cost, and carbon footprint by optimizing the power delivery infrastructure itself, while continuing to provide a highly reliable supply of power, e.g., using energy storage technologies [13, 14, 32] or designing flexible power switching or routing techniques [10, 24]. In contrast, we target a promising alternative approach: relaxing the requirement for a continuous power source and then designing high availability techniques to ensure software services remain available during unexpected power outages or shortages. We envision data centers with a heterogeneous power delivery infrastructure that includes a mix of servers connected to power supplies with different levels of reliability and cost. While some servers may continue to use a highly reliable, but expensive, infrastructure that includes connections to multiple electric grids, high-capacity UPSs, and backup generators, others may connect to only a single grid and use cheaper lower-capacity UPSs, and still others may rely solely on renewables with little or no UPS energy buffer. As we discuss in Section 2, permitting even slight reductions in the reliability of the power supply has the potential to decrease a data center's cost and carbon footprint.

To maintain server availability while also allowing data centers to "pull the plug" on servers if the level of available power suddenly changes, i.e., is no longer sufficient to power the active set of of servers, we introduce the abstraction of a *transient server*. A transient server's defining characteristic is that it may terminate anytime after an *advance warning* period. Transient servers arise in many scenarios. For example, spot instances in Amazon's Elastic Compute Cloud (EC2) terminate after a brief warning if the spot price ever exceeds the instance's bid price. As another example, UPSs built into racks provide some time after a power outage before servers completely lose power. In this paper, we apply the abstraction to green data centers that use renewables to power a fraction of servers, where the length of the warning period is a function of UPS capacity or expected future energy availability from renewables. In this context, we show that transient servers are cheaper and greener to operate than *stable servers*, which assume continuous power,

since they i) do not require an expensive power infrastructure that ensures 24x7 power availability and ii) can directly use intermittent renewable energy.

Unfortunately, transient servers expose applications to volatile changes in their server allotment, which degrade performance. Ideally, applications would always use as many transient servers as possible, but seamlessly transition to stable servers whenever transient servers become unavailable. To achieve this ideal, we propose system support for transient servers, called Yank, that maintains "live" backups of transient virtual machines' (VMs') memory and disk state on one or more stable *backup servers*. Yank extends the concept of whole system replication popularized by Remus [7] to exploit an advance warning period, enabling each backup server to support a many transient VMs. Highly multiplexing each backup server is critical to preserving transient servers' monetary and environmental benefits, allowing them to scale independently of the number of stable servers.

Importantly, the advance warning period eliminates the requirement that transient VMs always maintain external synchrony [23] with their backup to ensure correct operation, opening up the opportunity for both i) a looser form of synchrony and ii) multiple optimizations that increase performance and scalability. Our hypothesis is that a brief advance warning—on the order of a few seconds—enables Yank to provide high availability in the face of sudden changes in available power, cheaply and efficiently at large scales. In evaluating our hypothesis, we make the following contributions.

Transient Server Abstraction. We introduce the transient server abstraction, which supports a relaxed variant of external synchrony. Our variant, called just-in-time synchrony, exploits an advance warning that only ensures consistency with its backup before termination. We show how just-in-time synchrony applies to advance warnings with different durations, e.g., based on UPS capacity, and dynamics, e.g., based on intermittent renewables.

Performance Optimizations. We present multiple optimizations that further exploit the advance warning to scale the number of transient VMs each backup server supports without i) degrading VM performance during normal operation, ii) causing long downtimes when transient servers terminate, and iii) consuming excessive network resources. Our optimizations leverage basic insights about memory usage to minimize the in-memory state each backup server must write to stable storage.

Implementation and Evaluation. We implement Yank inside the Xen hypervisor and evaluate its performance and scalability in a range of scenarios, including with different size UPSs and using renewable energy sources. Our experiments demonstrate that a backup server can concurrently support up to 15 transient VMs with minimal performance degradation using an advance warning as small as 10 seconds, even when running memory-intensive interactive web applications, which is a challenging application for whole system replication.

Technique	*Overhead*	*Extra Cost*	*Warning*
Live Migration	None	None	Lengthy
Yank	**Low**	**Low**	**Modest**
High Availability	High	High	None

Table 1: Yank has less overhead and cost than high availability, but requires less warning than live migration.

2 Motivation and Background

We first define the transient server abstraction before discussing Yank's use of the abstraction in green data centers that use intermittent renewable power sources.

Transient Server Abstraction. We assume a virtualized data center where applications run inside VMs on one of two types of physical servers: (i) always-on *stable servers* with a highly reliable power source and (ii) *transient servers* that may terminate anytime. Central to our work is the notion of an *advance warning*, which signals that a transient server will shutdown after a delay T_{warn}.

Once a transient server receives an advance warning, a data center must move any VMs (and their associated state) hosted on the transient server to a stable server to maintain their availability. Depending on T_{warn}'s duration, two solutions exist to transition a VM to a stable server. If T_{warn} is large, it may be possible to live migrate a VM from a transient to a stable server. VM migration requires copying the memory and disk (if necessary) state [6], so the approach is only feasible if T_{warn} is long enough to accommodate the transfer. Completion times for migration are dependent on a VM's memory and disk size, with prior work reporting times up to one minute for VMs with only 1GB memory and no disk state [1, 19].

An alternative approach is to employ a *high availability mechanism*, such as Remus [7, 22], which requires maintaining a live backup copy of each transient VM on a stable server. In this case, a VM transparently fails over to the stable server whenever its transient server terminates. While the approach supports warning times of zero, it requires the high runtime overhead of continuously propagating state changes from the VM to its backup. In some cases, memory-intensive, interactive workloads may experience 5X degradation in latency [7]. Supporting an advance warning of zero also imposes a high cost, requiring a backup server to keep VM memory snapshots resident in its own memory. In essence, supporting a warning time of zero requires a 1:1 ratio between transient and backup servers. Unfortunately, storing memory snapshots on disk is not an option, since it would further degrade performance by reducing the memory bandwidth ($\sim$3000MB/s) of primary VMs to the disk bandwidth ($<$100MB/s) of the backup server.

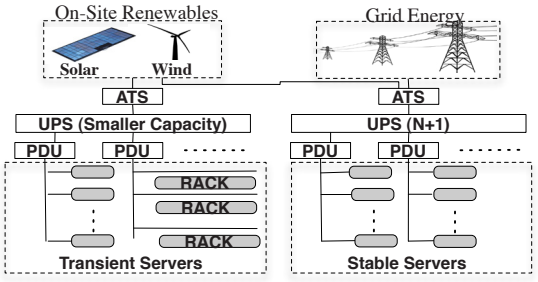


Figure 1: A data center with transient servers powered by renewables and low-capacity UPSs, and stable servers powered by the grid and redundant, high-capacity UPSs.

Live migration and high availability represent two extreme points in the design space for handling transient servers. The former has low overhead but requires long warning times, while the latter has high overhead but handles warning times of zero. Yank's goal is to exploit the middle ground between these two extremes, as outlined in Table 1, when a short advance warning is insufficient to complete a live migration, but does not necessarily warrant the overhead of externally synchronous live backups of VM memory and disk state. Yank optimizes for modest advance warnings by maintaining a backup copy, similar to Remus, of each transient VM's memory and disk state on a stable backup server. However, Yank focuses on keeping costs low, by highly multiplexing each backup server across many transient VMs.

As we discuss, our approach requires storing portions of each VM's memory backup on stable storage. We show that for advance warnings of a few seconds, Yank provides similar failover properties as high availability, but with an overhead and cost closer to live migration. In fact, Yank devolves to high availability for a warning time of zero, and reduces to a simple live migration as the warning time becomes larger. Yank focuses on scenarios where there is an advance warning of a fail-stop failure. Many of these failures stem from power outages where energy storage provides the requisite warning.

Green Data Center Model. Figure 1 depicts the data center infrastructure we assume in our work. As shown, the data center powers servers using two sources: (i) on-site renewables, such as solar and wind energy, and (ii) the electric grid. Recent work proposes a similar architecture for integrating renewables into data centers [18].

We assume that the on-site renewable sources power a significant fraction of the data center's servers. However, since renewable generation varies based on environmental conditions, this fraction also varies over time. While UPSs are able to absorb short-term fluctuations in renewable generation, e.g, over time-scales of seconds to minutes caused by a passing cloud or a temporary drop in wind speed, long-term fluctuations require switching servers to grid power or temporarily deactivating them. A key assumption in our work is that it is feasible to switch some, *but not all*, servers from renewable sources to the grid to account for these power shortfalls.

The constraint of powering some, but not all, servers from the grid arises if a data center limits its peak power usage to reduce its electricity bill. Since peak power disproportionately affects the electric grid's capital and operational costs, utilities routinely impose a surcharge on large industrial power consumers based solely on their peak demand, e.g., the maximum average power consumed over a 30 minute rolling window [10, 13]. Thus, data centers can reduce their electricity bills by capping grid power draw. In addition, to scale renewable deployments, data centers will increasingly need to handle their power variations locally, e.g., by activating and deactivating servers, since i) relying on the grid to absorb variations is challenging if renewables contribute a large fraction ($\sim$20%) of grid power and ii) absorbing variations entirely using UPS energy storage is expensive [13]. In the former case, rapid variations from renewables could cause grid instability, since generators may not be agile enough to balance electricity's supply and demand.

Thus, in our model, green data centers employ both stable and transient servers. Both server types require UPSs to handle short-term power fluctuations. However, we expect transient servers to require only a few seconds of expensive UPS capacity to absorb short-term power fluctuations, while stable servers may require tens of minutes of capacity to permit time to spin-up and transition to backup generators in case of a power outage.

3 Yank Design

Since Yank targets modest-sized advance warnings on the order of a few seconds, it cannot simply migrate a transient VM to a stable server after receiving a warning. To support shorter warning times, one option is to maintain backup copies (or snapshots) of a VM's memory and disk state on a dedicated stable server, and then continuously update the copies as they change. In this case, if a transient VM terminates, this *backup server* can restart the VM using the latest memory and disk snapshot. A high availability technique must commit changes to a VM's memory and disk state to a backup server frequently enough to support a warning time of zero. However, supporting a warning time of zero necessitates a 1:1 ratio between transient and backup servers, eliminating transient servers' monetary and environmental benefits.

In contrast, Yank leverages the advance warning time to scale the number of transient servers independently of the number of backup servers by controlling when and how frequently transient VMs commit state updates to the backup server. In essence, the warning time T_{warn} limits the amount of data a VM can commit to its backup server after receiving a warning. Thus, during normal op-

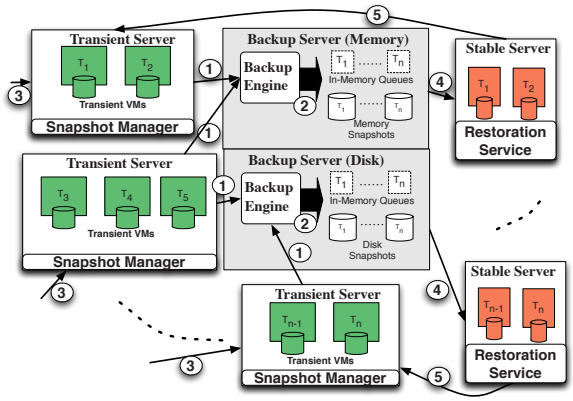


Figure 2: Yank's Design and Basic Operation

eration, a transient VM need only ensure the size of dirty memory pages and disk blocks remains below this limit. Maintaining this invariant guarantees that no update will be lost if a VM terminates after a warning, while providing additional flexibility over when to commit state. To keep its overhead and cost low, Yank highly multiplexes backup servers, allowing each to support many (>10) transient VMs by i) storing VM memory and disk snapshots, in part, on stable storage and ii) using multiple optimizations to prevent saturating disk bandwidth. Thus, given an advance warning, Yank supports the same failure properties as high availability, but uses fewer resources, e.g., hardware or power, for backup servers.

3.1 Yank Architecture

Figure 2 depicts Yank's architecture, which includes a *snapshot manager* on each transient server, a *backup engine* on each stable backup server, and a *restoration service* on each stable (non-backup) server. We focus primarily on how Yank maintains memory snapshots at the backup server, since we assume each backup server cannot keep memory snapshots for all transient VMs resident in its own memory. Thus, Yank must mask the order-of-magnitude difference between a transient VM's memory bandwidth (∼3000MB/s) and the backup server's disk bandwidth (< 100MB/s). By contrast, maintaining disk snapshots poses less of a performance concern, since the speed of a transient VM's disk and its backup server's disk are similar in magnitude. This characteristic combined with a multi-second warning time permits asynchronous disk mirroring to a backup server without significant performance degradation. Thus, while many of our optimizations below apply directly to disk snapshots, Yank currently uses off-the-shelf software (DRBD [9]) for disk mirroring.

Figure 2 also details Yank's functions. The snapshot manager executes within the hypervisor of each transient server and tracks the dirty memory pages of its resident VMs, periodically transmitting these pages to the backup engine, running at the backup server (1). The backup engine then queues each VM's dirty memory pages in its own memory before writing them to disk (2). Yank includes a service that monitors UPS state-of-charge, via the voltage level across its diodes, and translates the readings into a warning time based on the power consumption of transient servers (3). The service both i) informs backup and transient servers when the warning time changes and ii) issues warnings to transient and backup servers of an impending termination due to a power shortage. Since Yank depends on warning time estimates, the service above runs on a stable server. We discuss warning time estimation further in Section 3.5.

Upon receiving a warning (3), the snapshot manager pauses its VMs and commits any dirty pages to the backup engine before the transient server terminates. The backup engine then has two options, assuming it is too resource-constrained to run VMs itself: either store the VMs' memory images on disk for later use, or migrate the VMs to another stable (non-backup) server (4). Yank executes a simple restoration service on each stable (non-backup) server to facilitate rapid VM migration and restoration after a transient server terminates.

3.2 Just-in-Time Synchrony

Since Yank receives an advance warning of time T_{warn} before a transient server terminates, its VM memory snapshots stored on the backup server need not maintain strict external synchrony [23]. Instead, upon receiving a warning of impending termination, Yank only has to ensure what we call *just-in-time synchrony*: a transient VM and its memory snapshot on the backup server are always capable of being brought to a consistent state before termination. To guarantee just-in-time synchrony, as with external synchrony, the snapshot manager tracks dirty memory pages and transmits them to the backup engine, which then acknowledges their receipt. However, unlike external synchrony, just-in-time synchrony only *requires* the snapshot manager to buffer a VM's externally visible, e.g., network or disk, output when the size of the dirty memory pages exceed an upper threshold U_t, such that it is impossible to commit any more dirty pages to the backup engine within time T_{warn}.

In the worst case, with a VM that dirties pages faster than the backup engine is able to commit them, the snapshot manager is continually at the threshold, and Yank reverts to high availability-like behavior by always delaying the VM's externally visible output until its memory snapshot is consistent. Since we assume the backup server is not able to keep every transient VM memory snapshot resident in its own memory, the speed of the backup engine's disk limits the rate it is able to commit page changes. While memory bandwidth is an order of magnitude (or more) greater than disk (or network) bandwidth, Yank benefits from well-known characteristics of

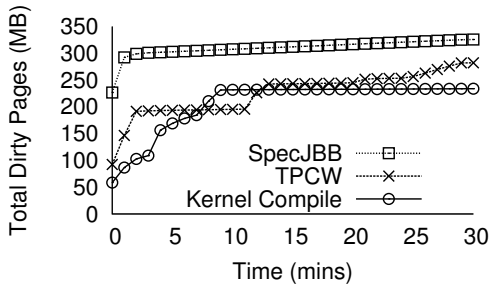


Figure 3: Working set size over time for three benchmarks: SPECjbb, TPCW, and Linux kernel compile.

typical in-memory application working sets to prevent saturating disk (or network) bandwidth. Specifically, the size of in-memory working sets tend to i) grow slowly over time and ii) be smaller than the total memory [8].

Slow growth stems from applications frequently re-writing the same memory pages, rather than always writing new ones. Yank only has to commit the last re-write of a dirty page (and not the intervening writes) to the backup server after reaching its upper threshold of dirty pages U_t. In contrast, to support termination with no advance warning, a VM must commit nearly *every* memory write to the backup server. In addition, small working sets enable the backup engine to keep most VMs' working sets in memory, reducing the likelihood of saturating disk bandwidth from writing dirty memory pages to disk. Recent work extends this insight to collections of VMs in data centers, showing that while the size of a single VM's working set may experience temporary bursts in memory usage, the bursts are often brief and not synchronized across VMs [33]. Yank relies on these observations in practice to highly multiplex each backup server's memory without saturating disk bandwidth or degrading transient VM performance during normal operation.

To confirm the characteristics above, we conducted a simple experiment: Figure 3 plots the dirty memory pages measured every 100ms for a VM with 1GB memory over a thirty minute period with three different applications: 1) the SPECjbb benchmark with 400 warehouses, 2) the TPC-W benchmark with 100 clients performing a browsing workload and 3) a Linux kernel (v 3.4) compile. The experiment verifies the observations above, namely that in each case i) the VM dirties less than 350MB or 35% of the available memory and ii) after experiencing an initial burst in memory usage the working set grows slowly over time.

Yank also relies on the observations above in setting its upper threshold U_t. To preserve just-in-time synchrony, this threshold represents the maximum size of the dirty pages the snapshot manager is capable of committing to the backup engine within the warning time. Our premise is that as long as the backup engine is able to keep each VM's working set in memory, even if all VMs simultaneously terminate, it should be able to commit any outstanding dirty pages without saturating disk bandwidth. Thus, U_t is a function of the warning time, the available network bandwidth between transient and backup servers, and the number of transient servers that may simultaneously terminate. For instance, for a single VM hosted on a transient server using a 1Gbps network link, a one second advance warning results in U_t=125MB. In Figure 3 above, U_t=125MB would only force SpecJBB to pause briefly on startup (where its usage briefly bursts above 125MB/s). The other applications never allocate more than 125MB in one second.

Of course, to support multiple VMs terminating simultaneously requires a lower U_t. However, as discussed in Section 3.5, Yank bases its warning time estimates on an "empty" UPS being at 40-50% depth-of-discharge. Thus, U_t need not be exactly precise, providing time to handle unlikely events, such as a warning coinciding with a correlated burst in VM memory usage, which may slow down commits by saturating the backup engine's disk bandwidth, or all VMs simultaneously terminating.

3.3 Optimizing VM Performance

For correctness, just-in-time synchrony only requires pausing a transient VM and committing dirty pages to the backup engine once their size reaches the upper threshold U_t, described above. A naïve approach only employs a single threshold at U_t by simply committing all dirty pages once their size reaches U_t. However, this approach has two drawbacks. First, it forces the VM to inevitably delay the release of externally visible output each time it reaches U_t, effectively pausing the VM from the perspective of external clients. If the warning time is long, e.g., 5-10 seconds, then the U_t will be large, causing long pauses. Second, it causes bursts in network traffic that affect other network applications.

To address these issues, the snapshot manager also uses a lower threshold L_t. Once the size of the dirty pages reaches L_t, it begins asynchronously committing dirty pages to the backup engine until the size is less than L_t. Algorithm 1 shows pseudo-code for the snapshot manager. The downside to using L_t is that the snapshot manager may end up committing more pages than with a single threshold, since it may unnecessarily commit the same memory page multiple times. Yank uses two techniques to mitigate this problem. First, to determine the order to commit pages, the snapshot manager associates a timestamp with each dirty page and uses a Least Recently Used (LRU) policy to prevent committing pages that are being frequently re-written. Second, the snapshot manager adaptively sets the lower threshold to be just higher than the VM's working set, since the working set contains the pages a VM is actively writing.

As we discuss in Section 5, our results show that as

Algorithm 1: Snapshot Manager's Algorithm for Committing Dirty Pages to the Backup Engine

```
Initialize Dirty Page Bitmap;
while No Warning Signal do
    Check Warning Time Estimate;
    Convert Warning Time to Upper Threshold (U_t);
    Get Num. Dirty Pages;
    if Num. Dirty Pages > U_t then
        Buffer Network Output of Transient VM;
        Transmit Dirty Pages to Backup Engine to
          Reduce Num. Dirty Pages to Lower Threshold (L_t);
        Wait for Ack. from Backup Engine;
        Unset Dirty Bitmap for Pages Sent;
        Release Buffered Network Packets;
    end
    if Num. Dirty Pages > L_t then
        Transmit Dirty Pages to Backup Engine at Specified Rate;
        Wait for Ack. from Backup Engine;
        Unset Dirty Bitmap for Pages Sent;
    end
end
Warning Received;
Pause the Transient VM;
Transmit Dirty Pages to Receiver Service;
Destroy the Transient VM;
```

long as the size of the VM's working set is less than U_t, using L_t results in smoother network traffic and fewer, shorter VM pauses. Of course, a combination of a large working set and short warning time may force Yank to degrade performance by continuously pausing the VM to commit frequently changing memory pages. In Section 5, we evaluate transient VM performance for a variety of applications with advance warnings in the range of 5-10 seconds. Finally, the snapshot manager implements standard optimizations to reduce network traffic, including content-based redundancy elimination and page deltas [34, 35]. The former technique associates memory pages with a hash based on their content, allowing the snapshot manager to send a 32b hash index rather than a 4kB memory page if the page is already present on the backup server. The technique is most useful in reducing the overhead of committing zero pages. The latter technique allows the snapshot manager to only send a page delta if it has previously committed a memory page.

3.4 Multiplexing the Backup Server

To multiplex many transient VMs, the backup engine must balance two competing objectives: using disk bandwidth efficiently during normal operation, while minimizing transient VM downtime in the event of a warning.

3.4.1 Maximizing Disk Efficiency

The backup engine maintains an in-memory queue for each transient VM to store newly committed (and acknowledged) memory pages. Since the backup server's memory is not large enough to store a complete memory snapshot for every transient VM it supports, it must inevitably write pages to disk. Yank includes multiple optimizations to prevent unnecessary disk writes.

First, when a transient VM commits a change to a memory page already present in its queue, the receiver deletes the out-of-date memory page without writing it to disk. As a result, the backup engine can often eliminate disk writes for frequently changing pages, even if the snapshot manager commits them. Second, to further prevent unnecessary writes, the backup engine orders each VM's queue using an LRU policy. Our use of LRU in both the snapshot manager (when determining which pages to commit on the transient VM) and the backup engine (when determining which pages to write to disk on the backup server) follows the same principles as a standard hierarchical cache. In addition, to exploit the observation that bursts in VM memory usage are not highly correlated, the backup engine selects pages to write to disk by applying LRU globally across all VM queues. Since it allocates a fixed amount of memory for all queues (near the backup server's total memory), the global LRU policy allows each VM's queue to grow and shrink as its working set size changes.

Finally, to further maximize disk efficiency, the backup engine could also use a log-structured file system [25], since its workload is write mostly, read rarely (only in the event of a failure). We discuss this design alternative and its implications in the next section.

3.4.2 Minimizing Downtime

The primary bottleneck in restoring a transient VM after a failure is the time required to read its memory state from the backup server's disk. Thus, to minimize downtime, as soon as a transient VM receives a warning, the backup engine immediately halts writing dirty pages to disk and begins reading the VM's existing (out-of-date) memory snapshot from disk, without synchronizing it with the queue of dirty page updates. Instead, the backup engine first sends the VM memory snapshot from disk to a restoration service, running on the destination stable (non-backup) server. We assume that an exogenous policy exists to select a destination stable server in advance to run a transient VM if its server terminates. In parallel, the backup engine also sends the VM's in-memory queue to the restoration service, after updating it with any outstanding dirty pages from the halted transient VM. To restore the VM, the restoration service applies the updates from the in-memory queue to the memory snapshot from the backup server's disk without writing to its own disk. Importantly, the sequence only requires reading a VM's memory snapshot from disk once, which begins as soon as the backup engine receives the warning. Note that if multiple transient VMs fail simultaneously, the backup engine reads and transmits memory snapshots from disk one at a time to maximize disk efficiency and minimize downtime across all VMs.

There are two design alternatives for determining how

the backup engine writes dirty page updates to its disk. As mentioned above, one approach is to use a log-structured file system. While a pure log-structured file system works well during normal operation, it results in random-access reads of a VM's memory snapshot stored on disk during failover, significantly increasing downtime (by ∼100X, the difference between random-access and sequential disk read bandwidth). In addition, maintaining log-structured files may lead to large log files on the backup server over time. The other alternative is to store each VM memory snapshot sequentially on disk, which results in slow random-access writes during normal operation but leads to smaller downtimes during failover because of faster sequential reads. Of course, this alternative is clearly preferable for solid state drives, since there is no difference between sequential and random write bandwidth. Considering these tradeoffs, in Yank's current implementation we use this design alternative to minimize downtime during failure.

3.5 Computing the Warning Time

Yank's correct operation depends on estimates of the advance warning time. There are multiple ways to compute the warning time. If transient servers connect to the grid, the warning time is static and based on server power consumption and UPS energy storage capacity. In the event of a power outage, if the UPS capacity is N watt-hours and the aggregate maximum server power is M watts, then the advance warning time is $\frac{N}{M}$. Alternatively, the warning time may vary in real-time if green data centers charge UPSs from on-site renewables. In this case, the UPSs' varying state-of-charge dictates the warning time, ensuring that transient servers are able to continue operation for some period if renewable generation immediately drops to zero. Note that warning time estimates do not need to be precisely accurate. Since, to minimize their amortized cost, data centers should not routinely use UPSs beyond a 40%-50% depth-of-discharge, even an "empty" UPS has some remaining charge to mind-the-gap and compensate for slight inaccuracies in warning time estimates. As a result, a natural buffer exists if warning time estimates are too short, e.g., by 1%-40%, although repeated inaccuracies degrade UPS lifetime.

4 Yank Implementation

Yank's implementation is available at http://yank.cs.umass.edu. We implement the snapshot manager by extending Remus inside the Xen hypervisor (v4.2). By default, Remus tracks dirty pages over short epochs (∼100ms) using shadow page tables and pausing VMs each epoch to copy dirty memory pages to a separate buffer for transmission to the backup server. While VMs may speculatively execute after copying dirty pages to the buffer, but before receiving

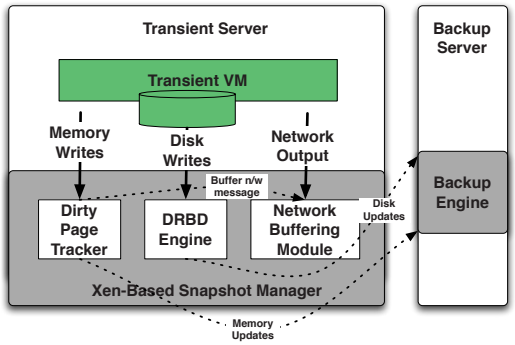


Figure 4: Snapshot Manager on the Transient Server

an acknowledgement from the backup server, they must buffer external network or disk output to preserve external synchrony. Remus only releases externally-visible output from these buffers after the backup server has acknowledged receiving dirty pages from the last epoch. Of course, by conforming to strict external synchrony, Remus enables a higher level of protection than Yank, including unexpected failures with no advance warning, e.g., fail-stop disk crashes. Although our current implementation only tracks dirty memory pages, it is straightforward to extend our approach to disk blocks.

Rather than commit dirty pages to the backup server every epoch, our snapshot manager uses a simple bitmap to track dirty pages and determine whether to commit these pages to the backup engine based on the upper and lower threshold, U_t and L_t. In addition, rather than commit CPU state each epoch, as in Remus, the snapshot manager only commits CPU state when it receives a warning that a transient server will terminate. Implementing the snapshot manager required adding or modifying roughly 600 lines of code (LOC), primarily in files related to VM migration, save/restore, and network buffering, e.g., xc_domain_save.c, xg_save_restore.h, and sch_plug.c. Finally, the snapshot manager includes a simple /proc interface to receive notifications about warnings or changes in the warning time. Figure 4 depicts the snapshot manager's implementation. As mentioned in the previous section, our current implementation uses DRBD to mirror disk state on a backup server.

Instead of modifying Xen, we implement Yank's backup engine "from scratch" at user-level for greater flexibility in controlling its in-memory queues and disk writing policy. The implementation is a combination of Python and C, with a Python front-end (∼300LOC) that accepts network connections and forks a backend C process (∼1500LOC) for each transient VM, as described below. Since the backup engine extends Xen's live migration and Remus functionality, the front-end listens on the same port (8002) that Xen uses for live migration. Figure 5 shows a detailed diagram of the backup engine.

For each transient VM, the backend C process accepts

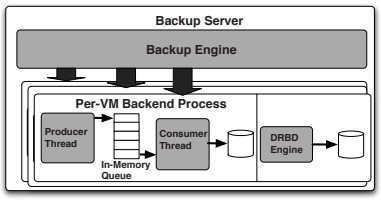


Figure 5: Backup Engine on the Backup Server

dirty page updates from the snapshot manager and sends acknowledgements. Each update includes the number of pages in the update, as well as each page's page number and contents (or a delta from the previous page sent). The process then places each update in an in-memory producer/consumer queue. To minimize disk writes, as described in Section 3.4.1, before queuing the update, the process checks the queue to see if a page already has a queued update. If so, the process merges the two updates. To perform these checks, the process maintains a hashtable that maps page numbers to their position in their queue. The process's consumer thread then removes pages from the queue (in LRU order based on a timestamp) and writes them to disk. In the current implementation, the backend process stores VM memory pages sequentially in a file on disk. For simplicity, the file's format is the same as Xen's format for storing saved VM memory images, e.g., via `xm save`. As discussed in Section 3.4.2, this results in low downtimes during migration, but lower performance during normal operation.

Finally, we implement Yank's restoration service (∼300LOC) at user-level in C. The daemon accepts a VM memory snapshot and an in-memory queue of pending updates, and then applies the updates to the snapshot without writing to disk. Since our implementation uses Xen's image format, the service uses `xm restore` from the resulting in-memory file to re-start the VM.

5 Experimental Evaluation

We evaluate Yank's network overhead, VM performance, downtime after a warning, and scalability, and then conduct case studies using real renewable energy sources. While our evaluation does not capture the full range of dynamics present in a production data center, it does demonstrate Yank's flexibility to handle a variety of dynamic and unexpected operating conditions.

5.1 Experimental Setup

We run our experiments on 20 blade servers with 2.13 GHz Xeon processors with 4GB of RAM connected to the same 1Gbps Ethernet switch. Each server running the snapshot manager uses our modified Xen (v4.2) hypervisor, while those running the backup engine and the restoration service use unmodified Xen (v4.2). In our experiments, each transient VM uses one CPU and 1GB RAM, and runs the same OS and kernel (Ubuntu 12.04, Linux kernel 3.2.0). We experiment with three benchmarks from Figure 3—TPC-W, SPECjbb, and a Linux kernel compile—to stress Yank in different ways.

TPC-W is a benchmark web application that emulates an online store akin to Amazon. We use a Java servlets-based multi-tiered configuration of TPC-W that uses Apache Tomcat (v7.0.27) as a front end and MySQL (v5.0.96) as a database backend. We use additional VMs to run clients that connect to the TPC-W shopping website. Our experiments use 100 clients connecting to TPC-W and performing the "browsing workload" where 95% of the clients only browse the website and the remaining 5% execute order transactions. TPC-W allows us to measure the influence of Yank on the response time perceived by the clients of an interactive web application.

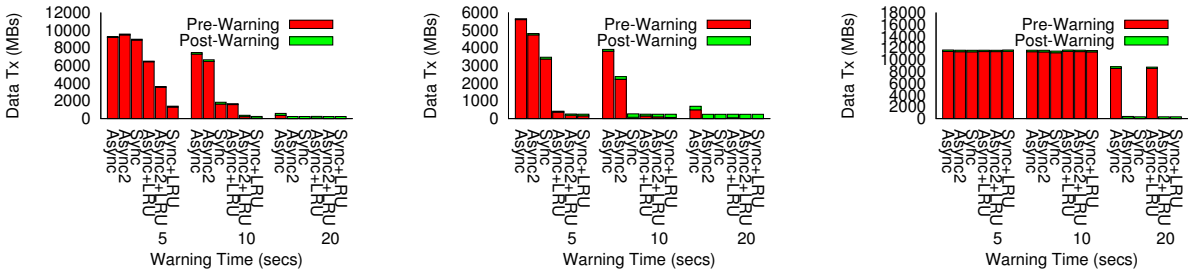
SPECjbb 2005 emulates a warehouse application for processing customer orders using a three-tier architecture comprising web, application, and database tiers. The benchmark predominantly exercises the middle tier that implements the business logic. We execute the benchmark on a single server in standalone mode using local application and database servers. SPECjbb is memory-intensive, dirtying memory at a fast rate, which stresses Yank's ability to maintain snapshots on the backup server without degrading VM performance.

Linux Kernel Compile compiles v3.5.3 of the kernel, along with all of its modules. The kernel compilation stresses both the memory and disk subsystems and is representative of a common development workload.

Note that the first two benchmarks are web applications, which are challenging due to their combination of interactivity and rapid writes to memory. We focus on interactive applications rather than non-interactive batch jobs, since the latter are more tolerant to delays and permit simple scheduling approaches to handling periodic power shortfalls, e.g., [2, 11, 12, 17]. Yank is applicable to batch jobs, although instead of scheduling them it shifts them to and from transient servers as power varies.

5.2 Benchmarking Yank's Performance

Network Overhead. Scaling Yank requires every transient VM to continuously send memory updates to the backup server. We first evaluate how much network traffic Yank generates, and how its optimizations help in reducing that traffic. As discussed in Section 3.3, the snapshot manager begins asynchronously committing dirty pages to the backup engine after reaching a lower threshold L_t. We compare this policy, which we call *async*, with the naïve policy, which we call *sync*, that enforces just-in-time synchrony by starting to commit dirty pages only after their size reaches the upper threshold U_t. Rather than commit all dirty pages when reaching U_t, which causes long pauses, the policy commits pages until the dirty pages reaches $0.9 * U_t$. We ex-



(a) Data Transferred with TPC-W (b) Data Transferred with Kernel Compile (c) Data Transferred with SPECjbb

Figure 6: Network overhead for each benchmark over a 15 minute period.

amine two variants of the async policy: the first one sets the lower threshold L_t to $0.5 * U_t$ and the second one sets it to $0.75 * U_t$. Our standard async and sync policies use a FIFO queue to select pages to commit to the backup engine; we label Yank's LRU optimization separately.

In this experiment, transient VMs draw power from the grid, and have a static warning time dictated by their UPS capacity. We also limit the backup engine to using an in-memory queue of 300MB to store memory updates from each 1GB transient VM. We run each experiment for 15 minutes before simulating a power outage by issuing a warning to the transient and backup server. We then measure the data transferred both before and after the warning for each benchmark. Figure 6 shows the results, which demonstrate that network usage, in terms of total data transferred, decreases with increasing warning time. As expected, the sync policy leads to less network usage than either async policy, since it only commits dirty pages when absolutely necessary. However, the experiment also shows that combining LRU with async reduces the network usage compared to async with a FIFO policy. We see that with just a 10 second warning time, Yank sends less than 100MB of data over 15 minutes for both TPC-W and the kernel compile, largely because after their initial memory burst these applications rewrite the same memory pages. For the memory-intensive SPECjbb benchmark, a 10 second warning time results in poor performance and excessive network usage. However, a 20 second warning time reduces network usage to $<$100MB over the 15 minute period.

Result: *The sync policy has the lowest network overhead, although async with LRU also results in low network overhead. With these policies, a 10 second warning leads to $<$ 100MB of data transfer over 15 minutes.*

Transient VM Performance. We evaluate Yank's effect on VM performance during normal operation by using the same experimental setup as before except that we do not issue any warning and only evaluate pre-warning performance. Here, we focus on TPC-W, since it is an interactive application that is sensitive to VM pauses from buffering network or disk output. We measure the average response time of TPC-W clients, while varying

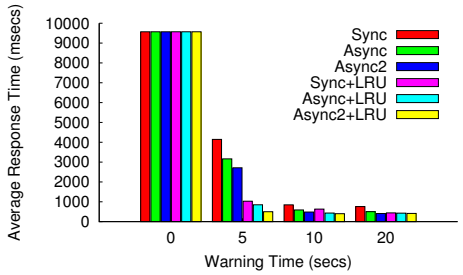


Figure 7: TPC-W response time as warning time varies.

both the warning time and the snapshot manager's policy for committing pages to the backup engine. Figure 7 shows that the async policy that selects pages to commit using and LRU policy results in the lowest average response time. The async policy reduces VM pauses, because the snapshot manager begins committing pages to the backup as soon as it hits the lower threshold L_t rather than waiting until reaching U_t, and forcing a large commit to the backup server. The experiment also demonstrates that with even a brief five second warning, the response time is $<$500ms using the async policy with LRU.

By contrast, with a warning time of zero the average response time rises to over nine seconds. In addition, the 90^{th} percentile response time was also near 15 seconds, indicating that the average response is not the result of a few overly bad response times. With no warning, the VM must pause and mirror every memory write to the backup server and receive an acknowledgement before proceeding. Although Remus [7] does not use our specific TPC-W benchmark, their results with the SPECweb benchmark are qualitatively similar, showing 5X worse latency scores. Thus, our results confirm that even modest advance warning times lead to vast improvements in response time for interactive applications.

Result: *Yank imposes minimal overhead on TPC-W during normal operation. With a brief five second warning time, the average response time of TPC-W is 10x less ($<$500ms) than with no warning time ($>$9s).*

VM Downtime after a Warning. The experiments above demonstrate that Yank imposes modest network and VM overhead during normal operation. In this experiment, we issue a warning to the transient server at

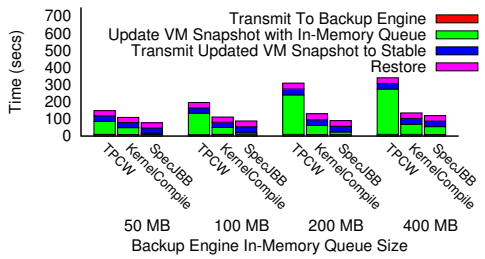


Figure 8: Downtime with the straightforward approach.

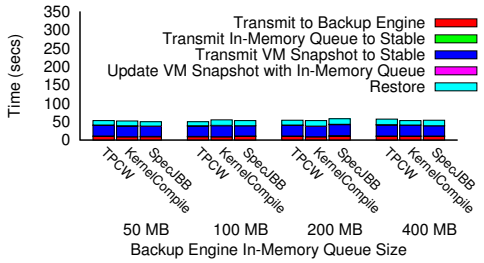


Figure 9: Downtime with Yank's optimizations.

the end of 15 minutes and measure downtime while the backup engine migrates the transient VM to a stable server. We compare Yank's approach (described in Section 3.4.1), which requires only a single read of the VM's memory image from disk, with a straightforward approach where the backup engine applies all updates to the memory snapshot before migrating it to the destination stable server. Note that in this latter case there is no need for Yank's restoration service, since the backup engine can simply perform a Xen stop-and-copy migration of the consistent memory snapshot at the backup server.

Figure 8 plots the transient VM's downtime using the straightforward approach, and Figure 9 plots it using Yank's approach. Each graph decomposes the downtime into each stage of the migration. In Figure 8, the largest component is the time required to create a consistent memory snapshot on the backup engine by updating the memory snapshot on disk. In addition, we run the experiment with different sizes of the in-memory queue to show that downtime increases with queue size, since a larger queue size requires writing more updates to disk after a warning. While reading the VM's memory snapshot from disk and transmitting it to the destination stable server still dominates downtime using Yank's approach (Figure 9), it is less than half than with the straightforward approach and is independent of the queue size. Note that Yank's downtimes are in the tens of seconds and bounded by the time to read a memory snapshot from disk. While these downtimes are not long enough to break TCP connections, they are much longer than the millisecond-level downtimes seen by live migration.

Result: *Yank minimizes transient VM downtime after a warning to a single read of its memory snapshot from disk, which results in a 50s downtime for a 1GB VM.*

Scalability. The experiments above focus on performance with a single VM. We also evaluate how many transient VMs the backup engine is able to support concurrently, and the resulting impact on transient VM performance during normal operation. Again, we focus on the TPC-W benchmark, since it is most sensitive to VM pauses. In this case, our experiments last for 30 minutes using a warning time of 10 seconds, and scale the number of transient VMs running TPC-W connected to the same backup server. We measure CPU and memory usage on the backup server, as well as the average response time of the TPC-W clients. Figure 10 shows the results, including the maximum of the average response time across all transient VMs observed by the TPC-W clients, the CPU utilization on the backup server, and the backup engine's memory usage as a percentage of total memory. The figure demonstrates that, in this case, the backup server is capable of supporting as many as 15 transient VMs without the average client response time exceeding 700ms. Note that without using Yank the average response time for TPC-W clients running our workload is 300ms. In addition, even when supporting 15 VMs, the backup engine does not completely use its entire CPU or memory.

Result: *Yank is able to highly multiplex each backup server. Our experiments indicate that with a warning time of 10 seconds, a backup server can support at least 15 transient VMs running TPC-W with little performance degradation for even a challenging interactive workload.*

5.3 Case Studies

The previous experiments benchmark different aspects of Yank's performance. In this section, we use case studies to show how Yank might perform in a real data center using renewable energy sources. We use traces from our own solar panel and wind turbine deployments, which we have used in prior work [4, 26, 27, 29]. Note that in these experiments the warning time changes as renewable generation fluctuates, since we assume renewables charge a small UPS that powers the servers.

5.3.1 Adapting to Renewable Generation

Figure 11 shows the renewable power generation from compressing a 3-day energy harvesting trace. For these experiments, we assume the UPS capacity dictates a maximum warning time of 20 seconds, and that each server requires a maximum power of 300W. Our results are conservative, since we assume servers always draw their maximum power. In the trace, at t=60, power generation falls below the 300W the server requires, causing the battery to discharge and the warning time to decrease. At t=80, power generation rises above 300W, causing the warning time to increase. Figure 11 shows the instantaneous response time of a TPC-W client running on a transient VM as power varies. The response time rises when

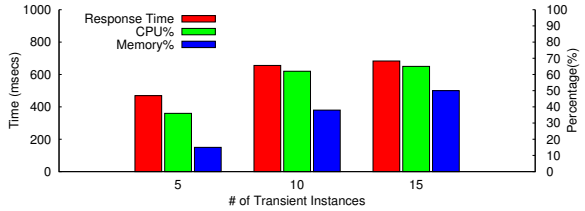


Figure 10: Yank scalability

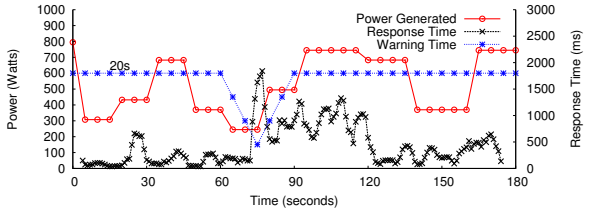


Figure 11: TPC-W response time as power varies

power generation decreases, and falls when it increases.

The experiment illustrates an important property of Yank: it is capable of translating variations in power availability to variations in application performance, *even for interactive application running on a single server*. Since servers remain far from energy-proportional, decreases in power availability require data centers to deactivate servers. Until now, the only way to scale application performance with power was to approximate energy proportionality in large clusters by deactivating a subset of servers [30]. For interactive applications not tolerant to delays, this approach is not ideal, especially if applications run on small set of servers. Of course, Yank's approach complements admission control policies that may simply reject clients to decrease load, rather than satisfying existing clients with a slightly longer response time. In many cases, simply rejecting new or existing clients may be undesirable.

Result: *Yank translates variations in power availability to variations in application performance for interactive applications running on a small number of servers.*

5.3.2 End-to-End Examples

We use end-to-end examples to demonstrate how Yank reacts to changes in renewable power by migrating transient VMs between transient and stable servers.

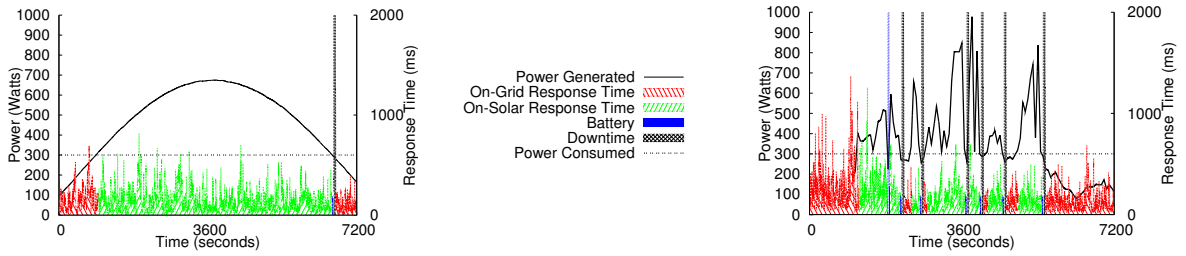
Solar Power. We first compress solar power traces from 7am to 5pm on both a sunny day and a cloudy day to a 2-hour period, and then scale the power level such that the trace's average power is equal to a server's maximum power (300W). While we compress our traces to enable experiments to finish within reasonable times, we do not introduce any artificial variability in the power generation, since renewable generation is already highly variable [31]. We then emulate a solar-powered transient server using a UPS that provides a maximum warning time of 30 seconds, although as above when power generation falls below 300W the UPS discharges and the warning time decreases. When the warning time reaches zero, Yank issues a warning and transfers the transient VM to a stable server. Likewise, when the warning time is non-zero continuously for a minute, Yank reactivates the transient server and transfers the VM back to it. Again, we run TPC-W in the VM and measure response time at the client as power generation varies.

Figure 12(a) and (b) shows that for both days Yank adapts to power variations with negligible impact on application performance. The sunny day (a) only requires the transient server to deactivate once, and has negligible impact on response time throughout the day. The cloudy day (b) requires the transient server to deactivate just seven times throughout the day. Thus, the application experiences seven brief periods of downtime, roughly 50 seconds in length, over the day. However, even though power is more intermittent than the sunny day, outside of these seven periods, the impact on response time is only slightly higher than during the sunny day. Note that, in this experiment, the periods where the VM executes on a stable server are brief, with Yank migrating the VM back to the transient server after a short time.

Wind Power. Wind power varies significantly more than solar power. Thus, in this experiment, we use a less conservative approach to computing the warning time. Rather than computing the warning time based on a UPS's remaining energy, we use a simple past-predicts-future (PPF) model (from [27]) to estimate future energy harvesting. The model predicts energy harvesting over the next 10 seconds will be same the same as the last 10 seconds. As above, we compress a wind energy trace from 7am to 5pm to two hours and scale its average power generation to the server's power. Since our PPF predictions operate over 10 second intervals, we use a UPS capacity that provides 10 seconds of power if predictions are wrong. We again measure TPC-W response time as it shifts between a transient and stable server.

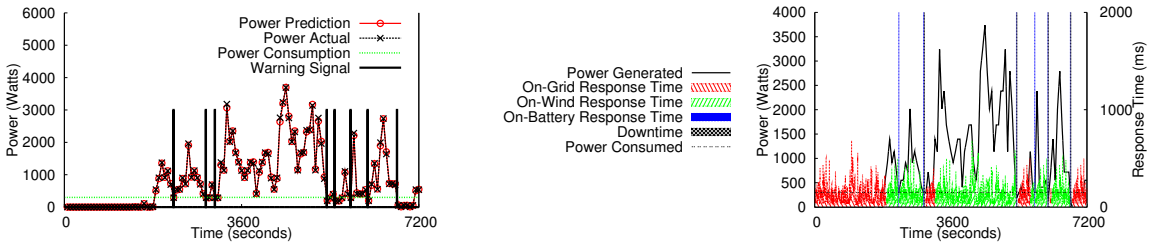
Figure 13(a) shows the PPF model accurately predicts power over these short timescales even though power generation varies significantly, allowing Yank to issue warnings with only a small amount of UPS power. Figure 13(b) shows the corresponding TPC-W response time, which is similar to the response time in the more stable solar traces. Of course, as during the cloudy day with solar power, when wind generation drops for a long period there is a brief downtime as the VM migrates from the transient server to a stable server.

Result: *Yank is flexible enough to handle different levels of intermittency in available power resulting from variations in renewable power generation.*



(a) Sunny Day (b) Cloudy Day

Figure 12: Yank using solar power on both a sunny and cloudy day.



(a) Short-term PPF prediction model (b) Wind-powered Server

Figure 13: Yank using wind power.

6 Related Work

Prior work on supporting renewables within a data center primarily targets non-interactive batch jobs, since these jobs are more tolerant to long delays when renewable power is not available. For example, GreenSlot [11] is a general batch job scheduler that uses predictions of future energy harvesting to align job execution with periods of high renewable generation. Similarly, related work [2, 12] proposes similar types of scheduling algorithms that specifically target MapReduce jobs. These solutions only support non-interactive batch jobs, and, in some cases, specifically target solar power [10], which is more predictable than wind power. Yank takes a different approach to support interactive applications running off renewable power. However, Yank's snapshots of memory and disk state are generic and also capable of supporting batch applications, although we leave a direct comparison of the two approaches to future work.

Recent work does combine interactive applications with renewables. For example, Krioukov et. al. design a power-proportional cluster targeting interactive applications that responds to power variations by simply deactivating servers and degrading request latency [17]. In prior work we propose a blinking abstraction for renewable-powered clusters, which we have applied to the distributed caches [26, 28] and distributed file systems [15] commonly used in data centers. However, blinking to support intermittent renewable energy requires significant application modifications, while Yank does not. iSwitch is perhaps the most closely-related work to Yank [18]. iSwitch assumes a similar design for integrating renewables into data centers, including some servers powered off renewables (specifically wind power) and some powered off the grid. However, iSwitch is more policy-oriented, tracking variations in renewable power to guide live VM migration between the two server pools. In contrast, Yank introduces a new mechanism, which iSwitch could use instead of live migration.

7 Conclusion

Yank introduces the abstraction of a transient server, which may terminate anytime after an advance warning. In this paper, we apply the abstraction to green data centers, where UPSs provides an advance warning, due to power shortfalls from renewables, move transient server state to stable servers. Yank fills the void between Remus, which requires no advance warning, and live VM migration, which requires a lengthy advance warning, to cheaply and efficiently support transient servers at large scale. In particular, our results show that a single backup server is capable of maintaining memory snapshots for up to 15 transient VMs with little performance degradation, which dramatically decreases the cost of providing high availability relative to existing solutions.

Acknowledgements. We would like to thank our shepherd, Ratul Mahajan, and the anonymous reviewers for their valuable comments, as well as Tim Wood and Navin Sharma for their feedback on early versions of this work. We also thank the Remus team, especially Shriram Rajagopalan, for assistance during early stages of this project. This research was supported by NSF grants CNS-0855128, CNS-0916972, OCI-1032765, CNS-1117221 and a gift from AT&T.

References

[1] S. Akoush, R. Sohan, A. Rice, A.W. Moore, and A. Hopper. Predicting the Performance of Virtual Machine Migration. In *MASCOTS*, August 2010.

[2] B. Aksanli, J. Venkatesh, L. Zhang, and T. Rosing. Utilizing Green Energy Prediction to Schedule Mixed Batch and Service Jobs in Data Centers. In *HotPower*, October 2011.

[3] Apple. Apple and the Environment. http://www.apple.com/environment/renewable-energy/, September 2012.

[4] S. Barker, A. Mishra, D. Irwin, E. Cecchet, P. Shenoy, and J. Albrecht. Smart*: An Open Data Set and Tools for Enabling Research in Sustainable Homes. In *SustKDD*, August 2012.

[5] L. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[6] C. Clark, K. Fraser, S. Hand, J. G Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *NSDI*, May 2005.

[7] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *NSDI*, April 2008.

[8] P. Denning. The Working Set Model for Program Behavior. *CACM*, 26(1), January 1983.

[9] DRBD. DRBD: Software Development for High Availability Clusters. http://www.drbd.org/, September 2012.

[10] I. Goiri, W. Katsak, K. Le, T. Nguyen, and R. Bianchini. Parasol and GreenSwitch: Managing Datacenters Powered by Renewable Energy. In *ASPLOS*, March 2013.

[11] I. Goiri, K. Le, M. Haque, R. Beauchea, T. Nguyen, J. Guitart, J. Torres, and R. Bianchini. GreenSlot: Scheduling Energy Consumption in Green Datacenters. In *SC*, April 2011.

[12] I. Goiri, K. Le, T. Nguyen, J. Guitart, J. Torres, and R. Bianchini. GreenHadoop: Leveraging Green Energy in Data-Processing Frameworks. In *EuroSys*, April 2012.

[13] S. Govindan, A. Sivasubramaniam, and B. Urgaonkar. Benefits and Limitations of Tapping into Stored Energy for Datacenters. In *ISCA*, June 2011.

[14] S. Govindan, D. Wang, L. Chen, A. Sivasubramaniam, and B. Urgaonkar. Towards Realizing a Low Cost and Highly Available Datacenter Power Infrastructure. In *HotPower*, October 2011.

[15] D. Irwin, N. Sharma, and P. Shenoy. Towards Continuous Policy-driven Demand Response in Data Centers. *Computer Communications Review*, 41(4), October 2011.

[16] Jonathan Koomey. Growth in Data Center Electricity Use 2005 to 2010. In *Analytics Press*, Oakland, California, August 2011.

[17] A. Krioukov, S. Alspaugh, P. Mohan, S. Dawson-Haggerty, D. E. Culler, and R. H. Katz. Design and Evaluation of an Energy Agile Computing Cluster. In *Technical Report UCB/EECS-2012-13, EECS Department, University of California, Berkeley*, January 2012.

[18] C. Li, A. Qouneh, and T. Li. iSwitch: Coordinating and Optimizing Renewable Energy Powered Server Clusters. In *ISCA*, June 2012.

[19] H. Liu, C. Xu, H. Jin, J. Gong, and X. Liao. Performance and Energy Modeling for Live Migration of Virtual Machines. In *HPDC*, June 2011.

[20] Microsoft. Becoming Carbon Nneutral: How Microsoft is Striving to Become Leaner, Greener, and More Accountable. *Microsoft*, June 2012.

[21] R. Miller. Facebook Installs Solar Panels at New Data Center. In *Data Center Knowledge*, April 16 2011.

[22] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield. RemusDB: Transparent High Availability for Database Systems. In *VLDB*, August 2011.

[23] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the Sync. In *OSDI*, November 2006.

[24] S. Pelley, D. Meisner, P. Zandevakili, T. Wenisch, and J. Underwood. Power Routing: Dynamic Power Provisioning in the Data Center. In *ASPLOS*, March 2010.

[25] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *TOCS*, 10(1), February 1992.

[26] N. Sharma, S. Barker, D. Irwin, and P. Shenoy. Blink: Managing Server Clusters on Intermittent Power. In *ASPLOS*, March 2011.

[27] N. Sharma, J. Gummeson, D. Irwin, and P. Shenoy. Cloudy Computing: Leveraging Weather Forecasts in Energy Harvesting Sensor Systems. In *SECON*, June 2010.

[28] N. Sharma, D. Krishnappa, D. Irwin, M. Zink, and P. Shenoy. GreenCache: Augmenting Off-the-Grid Cellular Towers with Multimedia Caches. In *MMSys*, February 2013.

[29] N. Sharma, P. Sharma, D. Irwin, and P. Shenoy. Predicting Solar Generation from Weather Forecasts Using Machine Learning. In *SmartGridComm*, October 2011.

[30] N. Tolia, Z. Wang, M. Marwah, C. Bash, P. Ranganathan, and X. Zhu. Delivering Energy Proportionality with Non Energy-Proportional Systems – Optimizing the Ensemble. In *HotPower*, December 2008.

[31] Y.H. Wan. Long-term Wind Power Variability. Technical report, National Renewable Energy Laboratory, January 2012.

[32] D. Wang, C. Ren, A. Sivasubramaniam, B. Urgaonkar, and H. Fathy. Energy Storage in Datacenters: What, Where, and How Much? In *SIGMETRICS*, June 2012.

[33] D. Williams, H. Jamjoom, Y. Liu, and H. Weatherspoon. Overdriver: Handling Memory Overload in an Oversubscribed Cloud. In *VEE*, 2011.

[34] T. Wood, A. Lagar-Cavilla, K. K. Ramakrishnan, P. Shenoy, and J. Van der Merwe. PipeCloud: Using Causality to Overcome Speed-of-Light Delays in Cloud-Based Disaster Recovery. In *SOCC*, October 2011.

[35] T. Wood, K. Ramakrishnan, P. Shenoy, and J. Van der Merwe. CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines. In *VEE*, March 2011.

Scalable Rule Management for Data Centers

Masoud Moshref[†] *Minlan Yu*[†] *Abhishek Sharma*[†*] *Ramesh Govindan*[†]
[†] *University of Southern California* [*] *NEC Labs America*

Abstract

Cloud operators increasingly need more and more fine-grained rules to better control individual network flows for various traffic management policies. In this paper, we explore automated rule management in the context of a system called vCRIB (a virtual Cloud Rule Information Base), which provides the abstraction of a centralized rule repository. The challenge in our approach is the design of algorithms that automatically off-load rule processing to overcome resource constraints on hypervisors and/or switches, while minimizing redirection traffic overhead and responding to system dynamics. vCRIB contains novel algorithms for finding feasible rule placements and adapting traffic overhead induced by rule placement in the face of traffic changes and VM migration. We demonstrate that vCRIB can find feasible rule placements with less than 10% traffic overhead even in cases where the traffic-optimal rule placement may be infeasible with respect to hypervisor CPU or memory constraints.

1 Introduction

To improve network utilization, application performance, fairness and cloud security among tenants in multi-tenant data centers, recent research has proposed many novel traffic management policies [8, 32, 28, 17]. These policies require *fine-grained* per-VM, per-VM-pair, or per-flow rules. Given the scale of today's data centers, the total number of rules within a data center can be hundreds of thousands or even millions (Section 2). Given the expected scale in the number of rules, rule processing in future data centers can hit CPU or memory resource constraints at servers (resulting in fewer resources for revenue-generating tenant applications) and rule memory constraints at the cheap, energy-hungry switches.

In this paper, we argue that future data centers will require *automated rule management* in order to ensure rule placement that respects resource constraints, minimizes traffic overhead, and automatically adapts to dynamics. We describe the design and implementation of a virtual Cloud Rule Information Base (vCRIB), which provides the *abstraction* of a centralized rule repository, and automatically manages rule placement without operator or tenant intervention (Figure 1). vCRIB manages rules for different policies in an integrated fashion even in the presence of system dynamics such as traffic changes or VM migration, and is able to manage a variety of data center configurations in which rule processing may be constrained either to switches or servers or may be permitted on both types of devices, and where both CPU and memory constraints may co-exist.

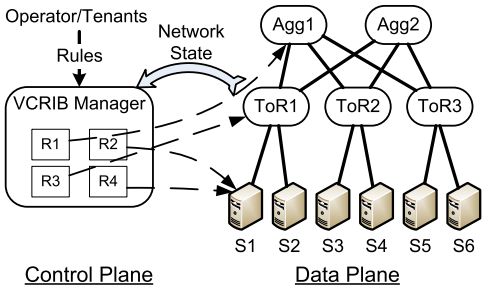


Figure 1: *Virtualized Cloud Rule Information Base (vCRIB)*

vCRIB's rule placement algorithms achieve resource-feasible, low-overhead rule placement by off-loading rule processing to nearby devices, thus trading off some traffic overhead to achieve resource feasibility. This trade-off is managed through a combination of three novel features (Section 3).

- Rule offloading is complicated by dependencies between rules caused by overlaps in the rule hyperspace. vCRIB uses per-source rule partitioning with replication, where the partitions encapsulate the dependencies, and replicating rules across partitions avoids rule inflation caused by splitting rules.
- vCRIB uses a *resource-aware placement* algorithm that offloads partitions to other devices in order to find a feasible placement of partitions, while also trying to co-locate partitions which share rules in order to optimize rule memory usage. This algorithm can deal with data center configurations in which some devices are constrained by memory and others by CPU.
- vCRIB also uses a *traffic-aware refinement algorithm* that can, either online, or in batch mode, refine partition placements to reduce traffic overhead while still preserving feasibility. This algorithm avoids local minima by defining novel benefit functions that perturb partitions allowing quicker convergence to feasi-

ble low overhead placement.

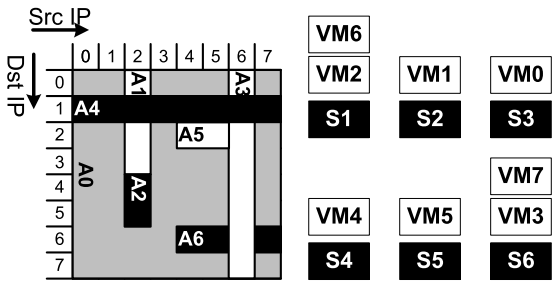
We evaluate (Section 4) vCRIB through large-scale simulations, as well as experiments on a prototype built on Open vSwitch [4] and POX [1]. Our results demonstrate that vCRIB is able to find feasible placements with a few percent traffic overhead, even for a particularly adversarial setting in which the current practice needs more memory than the memory capacity of all the servers combined. In this case, vCRIB is able to find a feasible placement, without relying on switch memory, albeit with about 20% traffic overhead; with modest amounts of switch memory, this overhead drops dramatically to less than 3%. Finally, vCRIB correctly handles heterogeneous resource constraints, imposes minimal additional traffic on core links, and converges within 5 seconds after VM migration or traffic changes.

2 Motivation and Challenges

Today, tenants in data centers operated by Amazon [5] or whose servers run software from VMware place their rules at the servers that source traffic. However, multiple tenants at a server may install too many rules at the same server causing unpredictable failures [2]. Rules consume resources at servers, which may otherwise be used for revenue-generating applications, while leaving many switch resources unused.

Motivated by this, we propose to automatically manage rules by offloading rule processing to other devices in the data center. The following paragraphs highlight the main design challenges in scalable automated rule management for data centers.

The need for many fine-grained rules. In this paper, we consider the class of data centers that provide computing as a service by allowing tenants to rent virtual machines (VMs). In this setting, tenants and data center operators need fine-grained control on VMs and flows to achieve different management *policies*. *Access control policies* either block unwanted traffic, or allocate resources to a group of traffic (e.g., rate limiting [32], fair sharing [29]). For example, to ensure each tenant gets a fair share of the bandwidth, Seawall [32] installs rules that match the source VM address and performs rate limiting on the corresponding flows. *Measurement policies* collect statistics of traffic at different places. For example, to enable customized routing for traffic engineering [8, 11] or energy efficiency [17], an operator may need to get traffic statistics using rules that match each flow (e.g., defined by five tuples) and count its number of bytes or packets. *Routing policies* customize the routing for some types of traffic. For example, Hedera [8] performs specific traffic engineering for large flows, while VLAN-based traffic management solutions [28] use different VLANs to route packets. Most of these policies,



(a) Wild card rules in a flow space (b) VM assignment

Figure 2: *Sample ruleset (black is accept, white is deny) and VM assignment (VM number is its IP)*

expressed in high level languages [18, 37], can be translated into virtual rules at switches[1].

A simple policy can result in a large number of fine-grained rules, especially when operators wish to control individual virtual machines and flows. For example, bandwidth allocation policies require one rule per VM pair [29] or per VM [29], and access control policies might require one rule per VM pair [30]. Data center traffic measurement studies have shown that 11% of server pairs in the same rack and 0.5% of inter-rack server pairs exchange traffic [22], so in a data center with $100K$ servers and 20 VMs per server, there can, be $1G$ to $20G$ rules in total ($200K$ per server) for access control or fair bandwidth allocation. Furthermore, state-of-the-art solutions for traffic engineering in data centers [8, 11, 17] are most effective when *per-flow* statistics are available. In today's data centers, switches routinely handle between 1K to 10K active flows within a one-second interval [10]. Assume a rack with 20 servers and if each server is the source of 50 to 500 active flows, then, for a data center with $100K$ servers, we can have up to $50M$ active flows, and need one measurement rule per-flow.

In addition, in a data center where multiple concurrent policies might co-exist, rules may have dependencies between them, so may require carefully designed offloading. For example, a rate-limiting rule at a source VM A can overlap with the access control rule that blocks traffic to destination VM B, because the packets from A to B match both rules. These rules cannot be offloaded to different devices.

Resource constraints. In modern data centers, rules can be processed either at servers (hypervisors) or programmable network switches (e.g., OpenFlow switches). Our focus in this paper is on flow-based rules that match packets on one or more header fields (e.g., IP addresses, MAC addresses, ports, VLAN tags) and perform various actions on the matching packets (e.g., drop, rate limit, count). Figure 2(a) shows a flow-space with source and

[1]Translating high-level policies to fine-grained rules is beyond the scope of our work.

destination IP dimensions (in practice, the flow space has 5 dimensions or more covering other packet header fields). We show seven flow-based rules in the space; for example, $A1$ represents a rule that blocks traffic from source IP 2 (VM2) to destination IP 0-3 (VM 0-3).

While software-based hypervisors at servers can support complex rules and actions (e.g., dynamically calculating rates of each flow [32]), they may require committing an entire core or a substantial fraction of a core at each server in the data center. Operators would prefer to allocate as much CPU/memory as possible to client VMs to maximize their revenue; e.g., RackSpace operators prefer not to dedicate even a portion of a server core for rule processing [6]. Some hypervisors offload rule processing to the NIC, which can only handle limited number of rules due to memory constraints. As a result, the number of rules the hypervisor can support is limited by the available CPU/memory budget for rule processing at the server.

We evaluate the numbers of rules and wildcard entries that can be supported by Open vSwitch, for different values of flow arrival rates and CPU budgets in Figure 3. With 50% of a core dedicated for rule processing and a flow arrival rate of 1K flows per second, the hypervisor can only support about 2K rules when there are 600 wildcard entries. This limit can easily be reached for some of the policies described above, so that manual placement of rules at sources can result in *infeasible* rule placement.

To achieve feasible placement, it may be necessary to offload rules from source hypervisors to other devices and redirect traffic to these devices. For instance, suppose VM2, and VM6 are located on S1 (Figure 2(b)). If the hypervisor at S1 does not have enough resources to process the deny rule $A3$ in Figure 2(a), we can install the rule at ToR1, introducing more traffic overhead. Indeed, some commercial products already support offloading rule processing from hypervisors to ToRs [7]. Similarly, if we were to install a measurement rule that counts traffic between S1 and S2 at Aggr1, it would cause the traffic between S1 and S2 to traverse through Aggr1 and then back. The central challenge is to design a collection of algorithms that manages this tradeoff — keeps the traffic overhead induced by rule offloading low, while respecting the resource constraint.

Offloading these rules to programmable switches, which leverage custom silicon to provide more scalable rule processing than hypervisors, is also subject to resource constraints. Handling the rules using expensive power-hungry TCAMs limits the switch capacity to a few thousand rules [15], and even if this number increases in the future its power and silicon usage limits its applicability. For example, the HP ProCurve 5406zl switch hardware can support about 1500 OpenFlow wildcard rules using TCAMs, and up to 64K Ethernet forwarding entries [15].

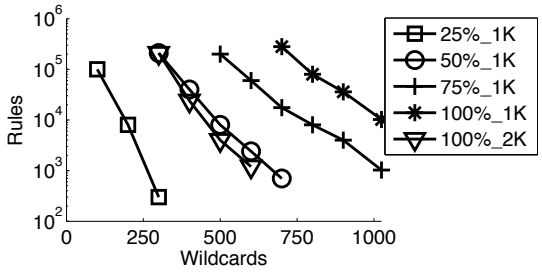


Figure 3: *Performance of openvswitch (The two numbers in the legend mean CPU usage of one core in percent and number of new flows per second.)*

Heterogeneity and dynamics. Rule management is further complicated by two other factors. Due to the different design tradeoffs between switches and hypervisors, in the future different data centers may choose to support either programmable switches, hypervisors, or even, especially in data centers with large rule bases, a combination of the two. Moreover, existing data centers may replace some existing devices with new models, resulting in device heterogeneity. Finding feasible placements with low traffic overhead in a large data center with different types of devices and qualitatively different constraints is a significant challenge. For example, in the topology of Figure 1, if rules were constrained by an operator to be only on servers, we would need to automatically determine whether to place a measurement rule for tenant traffic between S1 and S2 at one of those servers, but if the operator allowed rule placement at any device, we could choose between S1, ToR1, or S2; in either case, the tenant need not know the rule placement technology.

Today's data centers are highly dynamic environments with policy changes, VM migrations, and traffic changes. For example, if VM2 moves from $S1$ to $S3$, the rules $A0$, $A1$, $A2$ and $A4$ should me moved to $S3$ if there are enough resources at $S3$'s hypervisor. (This decision is complicated by the fact that $A4$ overlaps with $A3$.) When traffic changes, rules may need to be re-placed in order to satisfy resource constraints or reduce traffic overhead.

3 vCRIB Automated Rule Management

To address these challenges, we propose the design of a system called vCRIB (virtual Cloud Rule Information Base) (Figure 1). vCRIB provides the abstraction of a centralized repository of rules for the cloud. Tenants and operators simply install rules in this repository. Then vCRIB uses network state information including network topology and the traffic information to *proactively* place rules in hypervisors and/or switches in a way that respects resource constraints and minimizes the redirection traffic. Proactive rule placement incurs less controller overhead and lower data-path delays than a *purely reac-*

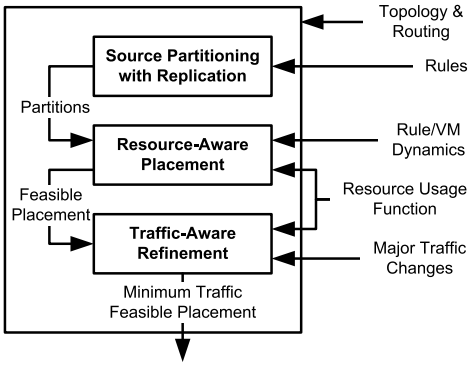


Figure 4: *vCRIB controller architecture*

tive approach, but needs sophisticated solutions to optimize placement and to quickly adapt to cloud dynamics (e.g., traffic changes and VM migrations), which is the subject of this paper. A hybrid approach, where some rules can be inserted reactively, is left to future work.

Designs \ Challenges	Overlapping rules	Resource constraints	Traffic overhead	Heterogeneity	Dynamics
Partitioning with replication	■	■			
Per-source partitions		■	■		
Similarity		■			
Resource usage functions				■	
Resource-aware placement		■			■
Traffic-aware refinement			■		■

Table 1: *Design choices and challenges mapping*

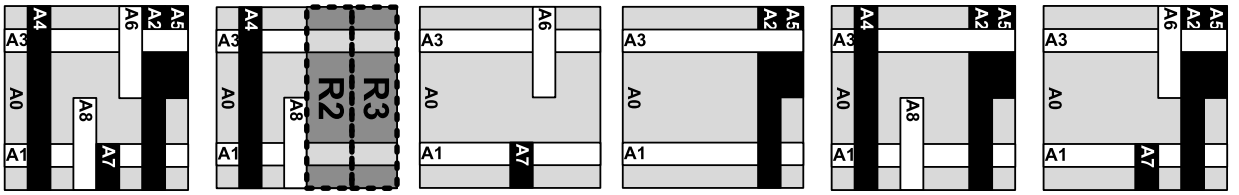
vCRIB makes several carefully chosen design decisions (Figure 4) that help address the diverse challenges discussed in Section 2 (Table 1). It partitions the rule space to break dependencies between rules, where each partition contains rules that can be co-located with each other; thus, a partition is the unit of offloading decisions. Rules that span multiple partitions are *replicated*, rather than split; this reduces rule inflation. vCRIB uses *per-source* partitions: within each partition, all rules have the same VM as the source so only a single rule is required to redirect traffic when that partition is offloaded. When there is *similarity* between co-located partitions (i.e., when partitions share rules), vCRIB is careful not to double resource usage (CPU/memory) for these rules, thereby scaling rule processing better. To accommodate device heterogeneity, vCRIB defines *resource usage functions* that deal with different constraints (CPU, memory etc.) in a uniform way. Finally, vCRIB splits the task of finding "good" partition off-loading opportunities into two steps: a novel bin-packing heuristic for *resource-aware partition placement* identifies feasible partition placements that respect resource constraints, and leverage similarity; and a fast *online traffic-aware refinement* algorithm which migrates partitions between devices to explore only feasible solutions while reducing traffic overhead. The split enables vCRIB to quickly adapt to small-scale dynamics (small traffic changes, or migration of a few VMs) without the need to recompute a feasible solution in some cases. These design decisions are discussed below in greater detail.

3.1 Rule Partitioning with Replication

The basic idea in vCRIB is to offload the rule processing from source hypervisors and allow more *flexible* and *efficient* placement of rules at both hypervisors and switches, while respecting resource constraints at devices and reducing the traffic overhead of offloading. Different types of rules may be best placed at different places. For instance, placing access control rules in the hypervisor (or at least at the ToR switches) can avoid injecting unwanted traffic into the network. In contrast, operations on the aggregates of traffic (e.g., measuring the traffic traversing the same link) can be easily performed at switches inside the network. Similarly, operations on inbound traffic from the Internet (e.g., load balancing) should be performed at the core/aggregate routers. Rate control is a task that can require cooperation between the hypervisors and the switches. Hypervisors can achieve end-to-end rate control by throttling individual flows or VMs [32], but in-network rate control can directly avoid buffer overflow at switches. Such flexibility can be used to manage resource constraints by moving rules to other devices.

However, rules cannot be moved unilaterally because there can be dependencies among them. Rules can overlap with each other especially when they are derived from different policies. For example, with respect to Figure 2, a flow from $VM6$ on server $S1$ to $VM1$ on server $S2$ matches both the rule $A3$ that blocks the source $VM1$ and the rule $A4$ that accepts traffic to destination $VM1$. When rules overlap, operators specify priorities so only the rule with the highest priority takes effect. For example, operators can set $A4$ to have higher priority. Overlapping rules make automated rule management more challenging because they constrain rule placement. For example, if we install $A3$ on S1 but $A4$ on ToR1, the traffic from $VM6$ to $VM1$, which should be accepted, matches $A3$ first and gets blocked.

One way to handle overlapping rules is to divide the flow space into multiple partitions and split the rule that intersects multiple partitions into multiple independent rules, *partition-with-splitting* [38]. Aggressive rule splitting can create many small partitions making it flexible to place the partitions at different switches [26], but can increase the number of rules, resulting in inflation. To minimize splitting, one can define a few large partitions, but these may reduce placement flexibility, since some partitions may not "fit" on some of the devices.



(a) Ruleset (b) Partition-with-replication (c) P1 & P3 on a device (d) P2 & P3 on a device

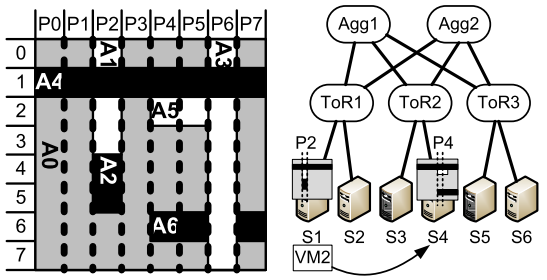
Figure 5: *Illustration of partition-with-replications (black is accept, white is deny)*

To achieve the flexibility of small partitions while limiting the effect of rule inflation, we propose a *partition-with-replication* approach that replicates the rules across multiple partitions instead of splitting them. Thus, in our approach, each partition contains the original rules that are covered partially or completely by that partition; these rules are not modified (e.g., by splitting). For example, considering the rule set in Figure 5(a), we can form the three partitions shown in Figure 5(b). We include both $A1$ and $A3$ in $P1$, the left one, in their original shape. The problem is that there are other rules (e.g., $A2$, $A7$) that overlap with $A1$ and $A3$, so if a packet matches $A1$ at the device where $P1$ is installed, it may take the wrong action – $A1$'s action instead of $A7$'s or $A2$'s action. To address this problem, we leverage redirection rules $R2$ or $R3$ at the source of the packet to completely cover the flow space of $P2$ or $P3$, respectively. In this way, any packets that are outside $P1$'s scope will match the redirection rules and get directed to the current host of the right partition where the packet can match the right rule. Notice that the other alternatives described above also require the same number of redirection rules, but we leverage high priority of the redirection rules to avoid incorrect matches.

Partition-with-replication allows vCRIB to flexibly manage partitions without rule inflation. For example, in Figure 5(c), we can place partitions $P1$ and $P3$ on one device; the same as in an approach that uses small partitions with rule splitting. The difference is that since $P1$ and $P3$ both have rules $A1$, $A3$ and $A0$, we only need to store 7 rules using partition-with-replication instead of 10 rules using small partitions. On the other hand, we can prove that the total number of rules using partition-with-replication is the same as placing one large partition per device with rule splitting (proof omitted for brevity).

vCRIB generates *per-source* partitions by cutting the flow space based on the source field according to the source IP addresses of each virtual machine. For example, Figure 6(a) presents eight per-source partitions $P0, \cdots, P7$ in the flow space separated by the dotted black lines.

Per-source partitions contain rules for traffic sourced by a single VM. Per-source partitions make the placement and refinement steps simpler. vCRIB only needs



(a) Per-source partitions (b) partition assignment

Figure 6: *Rule partition example*

one redirection rule installed at the source hypervisor to direct the traffic to the place where the partition is stored. Unlike per-source partitions, a partition that spans multiple source may need to be replicated; vCRIB does not need to replicate partitions. Partitions are ordered in the source dimension, making it easy to identify similar partitions to place on the same device.

3.2 Partition Assignment and Resource Usage

The central challenge in vCRIB design is the assignment of partitions to devices. In general, we can formulate this as an optimization problem, whose goal is to minimize the total traffic overhead subject to the resource constraints at each device.[2] This problem, even for partition-with-splitting, is equivalent to the *generalized assignment problem*, which is NP-hard and even APX-hard to approximate [14]. Moreover, existing approximation algorithms for this problem are inefficient. We refer the reader to a technical report which discusses this in greater depth [27].

We propose a two-step heuristic algorithm to solve this problem. First, we perform *resource-aware placement* of partitions, a step which only considers resource constraints; next, we perform *traffic-aware refinement*, a step in which partitions reassigned from one device to another to reduce traffic overhead. An alternative approach might have mapped partitions to devices first to minimize traffic overhead (e.g., placing all the partitions at the source), and then refined the assignments to fit resource constraints. With this approach, however, we

[2] One may formulate other optimization problems such as minimizing the resource usage given the traffic usage budget. A similar greedy heuristic can also be devised for these settings.

cannot guarantee that we can find a feasible solution in the second stage. Similar two-step approaches have also been used in the resource-aware placement of VMs across servers [20]. However, placing partitions is more difficult than placing VMs because it is important to co-locate partitions which share rules, and placing partitions at different devices incurs different resource usage.

Before discussing these algorithms, we describe how vCRIB models resource usage in hypervisors and switches in a uniform way. As discussed in Section 2, CPU and memory constraints at hypervisors and switches can impact rule placement decisions. We model resource constraints using a function $\mathscr{F}(P,d)$; specifically, $\mathscr{F}(P,d)$ is the percentage of the resource consumed by placing partition P on a device d. $\mathscr{F}$ determines how many rules a device can store, based on the rule patterns (i.e., exact match, prefix-based matching, and match based on wildcard ranges) and the resource constraints (i.e., CPU, memory). For example, for a *hardware OpenFlow switch* d with $s_{TCAM}(d)$ TCAM entries and $s_{SRAM}(d)$ SRAM entries, the resource consumption $\mathscr{F}(P,d) = r_e(P)/s_{SRAM}(d) + r_w(P)/s_{TCAM}(d)$, where r_e and r_w are the numbers of exact matching rules and wildcard rules in P respectively.

The resource function for *Open vSwitch* is more complicated and depends upon the number of rules $r(P)$ in the partition P, the number of wildcard patterns $w(P)$ in P, and the rate $k(d)$ of new flow arriving at switch d. Figure 3 shows the number of rules an Open vSwitch can support for different number of wild card patterns.[3] The number of rules it can support reduces exponentially with the increase of the number of wild card patterns (the y-axis in Figure 3 is in log-scale), because Open vSwitch creates a hash table for each wild card pattern and goes through these tables linearly. For a fixed number of wild card patterns and the number of rules, to double the number of new flows that Open vSwitch can support, we must double the CPU allocation.

We capture the CPU resource demand of Open vSwitch as a function of the number of new flows per second matching the rules in partition and the number of rules and wild card patterns handled by it. Using non-linear least squares regression, we achieved a good fit for Open vSwitch performance in Figure 3 with the function $\mathscr{F}(P,d) = \alpha(d) \times k(d) \times w(P) \times \log\left(\frac{\beta(d)r(P)}{w(P)}\right)$, where $\alpha = 1.3 \times 10^{-5}$, $\beta = 232$, with $R^2 = 0.95$.[4]

[3] The IP prefixes with different lengths 10.2.0.0/24 and 10.2.0.0/16 are two wildcard patterns. The number of wildcard patterns can be large when the rules are defined on multiple tuples. For example, the source and destination pairs can have at most 33*33 wildcard patterns.

[4] R^2 is a measure of *goodness of fit* with a value of 1 denoting a perfect fit.

3.3 Resource-aware Placement

Resource-aware partition placement where partitions do not have rules in common can be formulated as a bin-packing problem that minimizes the total number of devices to fit all the partitions. This bin-packing problem is NP-hard, but there exist approximation algorithms for it [21]. However, resource-aware partition placement for vCRIB is more challenging since partitions may have rules in common and it is important to co-locate partitions with shared rules in order to save resources.

Algorithm 1 First Fit Decreasing Similarity Algorithm

```
P = set of not placed partitions
while |P| > 0 do
    Select a partition P_i randomly
    Place P_i on an empty device M_k.
    repeat
        Select P_j ∈ P with maximum similarity to P_i
    until Placing P_j on M_k Fails
end while
```

We use a heuristic algorithm for bin-packing similar partitions called *First Fit Decreasing Similarity* (FFDS) (Algorithm 1) which extends the traditional FFD algorithm [33] for bin packing to consider *similarity* between partitions. One way to define similarity between two partitions is as the number of rules they share. For example, the similarity between $P4$ and $P5$ is $|P4 \cap P5| = |P4| + |P5| - |P4 \cup P5| = 4$. However, different devices may have different resource constraints (one may be constrained by CPU, and another by memory). A more general definition of similarity between partitions P_i and P_k on device d is based on the resource consumption function $\mathscr{F}$: our similarity function $\mathscr{F}(P_i,d) + \mathscr{F}(P_k,d) - \mathscr{F}(P_i \cup P_k,d)$ compares the network resource usage of co-locating those partitions.

Given this similarity definition, FFDS first picks a partition P_i randomly and stores it in a new device.[5] Next, we pick partitions similar to P_i until the device cannot fit more. Finally, we repeat the first step till we go through all the partitions.

For the memory usage model, since we use per-source partitions, we can quickly find partitions similar to a given partition, and improve the execution time of the algorithm from a few minutes to a second. Since per-source partitions are ordered in the source IP dimension and the rules are always contiguous blocks crossing only

[5] As a greedy algorithm, one would expect to pick large partitions first. However, since we have different resource functions for different devices, it is hard to pick the large partitions based on different metrics. Fortunately, in theory, picking partitions randomly or greedily do not affect the approximation bound of the algorithm. As an optimization, instead of picking a new device, we can pick the device whose existing rules are most similar to the new partition.

neighboring partitions, we can prove that the most similar partitions are always the ones adjacent to the partition [27]). For example, $P4$ has 4 common rules with $P5$ but 3 common rules with $P7$ in Figure 6(a). So in the third step of FFDS, we only need to compare left and right unassigned partitions.

To illustrate the algorithm, suppose each server in the topology of Figure 1 has a capacity of four rules to place the partitions and switches have no capacity. Considering the ruleset in Figure 2(a), we first pick a random partition $P4$ and place it on an empty device. Then, we check $P3$ and $P5$ and pick $P5$ as it has more similar rules (4 vs 2). Between $P3$ and $P6$, $P6$ is the most similar but the device has no additional capacity for $A3$, so we stop. In the next round, we place $P2$ on an empty device and bring $P1$, $P0$ and $P3$ but stop at $P6$ again. The last device will contain $P6$ and $P7$.

We have proved that, FFDS algorithm is 2-approximation for resource-aware placement in networks with only memory-constrained devices [27]. Approximation bounds for CPU-constrained devices is left to future work.

Our FFDS algorithm is inspired by the tree-based placement algorithm proposed in [33], which minimizes the number of servers to place VMs by putting VMs with more common memory pages together. There are three key differences: (1) since we use per-source partitions, it is easier to find the most similar partitions than memory pages; (2) instead of placing sub-trees of VMs in the same device, we place a set of similar partitions in the same device since these similar partitions are not bounded by the boundaries of a sub-tree; and (3) we are able to achieve a tighter approximation bound (2, instead of 3). (The construction of sub-trees is discussed in a technical report [27]).

Finally, it might seem that, because vCRIB uses per-source partitions, it cannot efficiently handle a rule with a wildcard on the source IP dimension. Such a rule would have to be placed in every partition in the source IP range specified by the wildcard. Interestingly, in this case vCRIB works quite well: since all partitions on a machine will have this rule, our similarity-based placement will result in only one copy of this rule per device.

3.4 Traffic-aware Refinement

The resource-aware placement places partitions without heed to traffic overhead since a partition may be placed in a device other than the source, but the resulting assignment is *feasible* in the sense that it respects resource constraints. We now describe an algorithm that *refines* this initial placement to reduce traffic overhead, while still maintaining feasibility. Having thus separated placement and refinement, we can run the (usually) fast refinement after small-scale dynamics (some kinds of traffic changes, VM migration, or rule changes) that do not violate resource feasibility. Because each per-source partition matches traffic from exactly one source, the refinement algorithm only stores each partition *once* in the entire network but tries to migrate it closer to its source.

Given per-source partitions, an *overhead-greedy* heuristic would repeatedly pick the partition with the largest traffic overhead, and place it on the device which has enough resources to store the partition and the lowest traffic overhead. However, this algorithm cannot handle dynamics, such as traffic changes or VM migration. This is because in the steady state many partitions are already in their best locations, making it hard to rearrange other partitions to reduce their traffic overhead. For example, in Figure 6(a), assume the traffic for each rule (excluding A0) is proportional to the area it covers and generated from servers in topology of Figure 6(b). Suppose each server has a capacity of 5 rules and we put $P4$ on $S4$ which is the source of $VM4$, so it imposes no traffic overhead. Now if $VM2$ migrates from $S1$ to $S4$, we cannot save both $P2$ and $P4$ on $S4$ as it will need space for 6 rules, so one of them must reside on ToR2. As $P2$ has 3 units deny traffic overhead on $A1$ plus 2 units of accept traffic overhead from local flows of $S4$, we need to bring $P4$ out of its sweet spot and put $P2$ instead. However, the overhead-greedy algorithm cannot move $P4$ as it is already in its best location.

To get around this problem, it is important to choose a potential refinement step that not only considers the benefit of moving the selected partition, but also considers the other partitions that might take its place in future refinement steps. We do this by calculating the *benefit* of moving a partition P_i from its current device $d(P_i)$ to a new device j, $M(P_i, j)$. The benefit comes from two parts: (1) The reduction in traffic (the first term of Equation 1); (2) The potential benefit of moving other partitions to $d(P_i)$ using the freed resources from P_i, excluding the lost benefit of moving these partitions to j because P_i takes the resources at j (the second term of Equation 1). We define the potential benefit of moving other partitions to a device j as the maximum benefits of moving a partition P_k from a device d to j, i.e., $Q_j = max_{k,d}(T(P_k,d) - T(P_k,j))$. We speed up the calculation of Q_j by only considering the current device of P_k and the best device $b(P_k)$ for P_k with the least traffic overhead. (We omit the reasons for brevity.) In summary, the benefit function is defined as:

$$M(P_i, j) = (T(P_i, d(P_i)) - T(P_i, j)) + (Q_{d(P_i)} - Q_j) \quad (1)$$

Our traffic-aware refinement algorithm is *benefit-greedy*, as described in Algorithm 2. The algorithm is given a time budget (a "timeout") to run; in practice, we

Algorithm 2 Benefit-Greedy algorithm

```
Update b(P_i) and Q(d)
while not timeout do
    Update the benefit of moving every P_i to its best feasible
    target device M(P_i, b(P_i))
    Select P_i with the largest benefit M(P_i, b(P_i))
    Select the target device j for P_i that maximizes the benefit
    M(P_i, j)
    Update best feasible target devices for partitions and Q's
end while
return the best solution found
```

have found time budgets of a few seconds to be sufficient to generate low traffic-overhead refinements. At each step, it first picks that partition P_i that would benefit the most by moving to its best feasible device $b(P_i)$, and then picks the most beneficial and feasible device j to move P_i to.[6]

We now illustrate the benefit-greedy algorithm (Algorithm 2) using our running example in Figure 6(b). The best feasible target device for both $P2$ and $P4$ are $ToR2$. $P2$ maximizes Q_{S4} with value 5 because its deny traffic is 3 and has 1 unit of accept traffic to $VM4$ on $S4$. Also we assume that Q_j is zero for all other devices. In the first step, the benefit of migrating $P2$ to $ToR2$ is larger than moving $P4$ to $ToR2$, while the benefits of all the other migration steps are negative. After moving $P2$ to $ToR2$ the only beneficial step is moving $P4$ out of $S4$. After moving $P4$ to $ToR2$, migrating $P2$ to $S4$ become feasible, so Q_{S4} will become 0 and as a result the benefit of this migration step will be 5. So the last step is moving $P2$ to $S4$.

An alternative to using a greedy approach would have been to devise a randomized algorithm for perturbing partitions. For example, a Markov approximation method is used in [20] for VM placement. In this approach, checking feasibility of a partition movement to create the links in the Markov chain turns out to be computationally expensive. Moreover, a randomized iterative refinement takes much longer to converge after a traffic change or a VM migration.

4 Evaluation

We first use simulations on a large fat-tree topology with many fine-grained rules to study vCRIB's ability to minimize traffic overhead given resource constraints. Next, we explore how the online benefit-greedy algorithm handles rule re-placement as a result of VM migrations. Our simulations are run on a machine with quad-core 3.4 GHz CPU and 16 GB Memory. Finally, we deploy our prototype in a small testbed to understand the overhead at the controller, and end-to-end delay between detecting traffic changes and re-installing the rules.

[6] By feasible device, we mean the device has enough resources to store the partition according to the function $\mathscr{F}$.

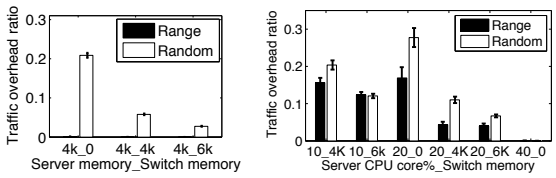
4.1 Simulation Setup

Topology: Our simulations use a three-level fat-tree topology with degree 16, containing 1024 servers in 128 racks connected by 320 switches. Since current hypervisor implementations can support multiple concurrent VMs [31], we use 20 VMs per machine. We consider two models of resource constraints at the servers: memory constraints (e.g., when rules are offloaded to a NIC), and CPU constraints (e.g., in Open vSwitch). For switches, we only consider memory constraints.

Rules: Since we do not have access to realistic data center rule bases, we use ClassBench [35] to create 200K synthetic rules each having 5 fields. ClassBench has been shown to generates rules representative of real-world access control.

VM IP address assignment: The IP address assigned to a VM determines the number of rules the VM matches. A random address assignment that is oblivious to the rules generated in the previous set may cause most of the traffic to match the default rule. Instead, we use a heuristic – we first segment the IP range with the boundaries of rules on the source and destination IP dimensions and pick random IP addresses from randomly chosen ranges. We test two arrangements: *Random* allocation which assigns these IPs randomly to servers and *Range* allocation which assigns a block of IPs to each server so the IP addresses of VMs on a server are in the same range.

Flow generation: Following prior work, we use a staggered traffic distribution (ToRP=0.5, PodP=0.3, CoreP=0.2) [8]. We assume that each machine has an average of 1K flows that are uniformly distributed among hosted VMs; this represents larger traffic than has been reported [10], and allows us to stress vCRIB. For each server, we select the source IP of a flow randomly from the VMs hosted on that machine and select the destination IP from one of the target machines matching the traffic distribution specified above. The protocol and port fields of flows also affect the distribution of used rules. The source port is wildcarded for ClassBench rules so we pick that randomly. We pick the destination port based on the protocol fields and the port distributions for different protocols (This helps us cover more rules and do not dwell on different port values for ICMP protocol.). Flow sizes are selected from a Pareto distribution [10]. Since CPU processing is impacted by newly arriving flows, we marked a subset of these flows as new flows in order to exercise the CPU resource constraint [10]. We run each experiment multiple times with different random seeds to get a stable mean and standard deviation.



(a) Memory budget at servers (b) CPU budget at servers

Figure 7: *Traffic overhead and resource constraints tradeoffs*

4.2 Resource Usage and Traffic Trade-off

The goal of vCRIB rule placement is to minimize the traffic overhead given the resource constraints. To calibrate vCRIB's performance, we compare it against *SourcePlacement*, which stores the rules at the source hypervisor. Our metric for the efficacy of vCRIB's performance is the ratio of traffic as a result of vCRIB's rule placement to the traffic incurred as a result of SourcePlacement (regardless of whether SourcePlacement is feasible or not). When *all* the servers have enough capacity to process rules (i.e., SourcePlacement is feasible), it incurs lowest traffic overhead; in these cases, vCRIB automatically picks the same rule placement as SourcePlacement, so here we only evaluate cases that SourcePlacement is infeasible. We begin with memory resource model at servers because of its simpler similarity model and later compare it with CPU-constrained servers.

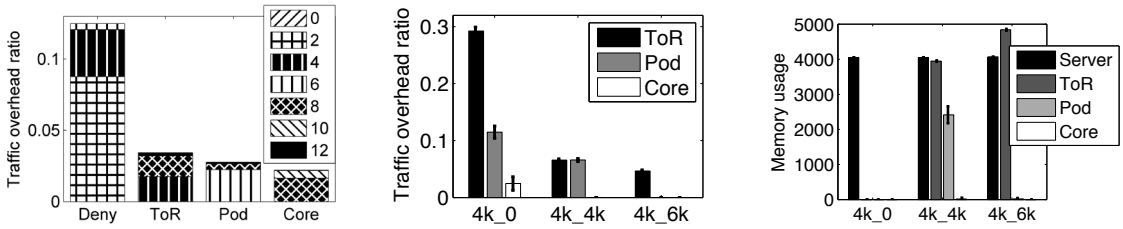
vCRIB uses similarity to find feasible solutions when SourcePlacement is infeasible. With *Range* IP allocation, partitions in the Source IP dimension which are similar to each other are saved on one server, so the average load on machines is smaller for SourcePlacement. However, there may still be a few overloaded machines that result in an infeasible SourcePlacement. With *Random* IP allocation, the partitions on a server have low similarity and as a result the average load of machines is larger and there are many overloaded ones. Having the maximum load of machines above 5K in all runs for both Range and Random cases, we set a capacity of 4K for servers and 0 for switches ("4K_0" setting) to make SourcePlacement infeasible. vCRIB could successfully fit all the rules in the servers by leveraging the similarities of partitions and balancing the rules. The power of leveraging similarity is evident when we observe that in the Random case *the average number of rules per machine (4.2K) for SourcePlacement exceeds the server capacity*, yet vCRIB finds a feasible placement by saving similar partitions on the same machine. Moreover, vCRIB finds a feasible solution when we add switch capacity and uses this capacity to optimize traffic (see below), yet SourcePlacement is unable to offload the load.

vCRIB finds a placement with low traffic overhead. Figure 7(a) shows the traffic ratio between vCRIB and SourcePlacement for the Range and Random cases with error bars representing standard deviation for 10 runs. For the Range IP assignment, vCRIB minimizes the traffic overhead under 0.1%. The worst-case traffic overhead for vCRIB is 21% when vCRIB cannot leverage rule processing in switches to place rules and the VM IP address allocation is random, an adversarial setting for vCRIB. The reason is that in the Random case the arrangement of the traffic sources is oblivious to the similarity of partitions. So any feasible placement depending on similarity puts partitions far from their sources and incurs traffic overhead. When it is possible to process rules on switches, *vCRIB's traffic overhead decreases dramatically* (6% (3%) for 4K (6K) rule capacity in internal switches); in these cases, to meet resource constraints, vCRIB places partitions on ToR switches on the path of traffic, incurring minimal overhead. As an aside, these results illustrate the potential for using vCRIB's algorithms for *provisioning*: a data center operator might decide when, and how much, to add switch rule processing resources by exploring the trade-off between traffic and resource usage.

vCRIB can also optimize placement given CPU constraints. We now consider the case where servers may be constrained by CPU allocated for rule processing (Figure 7(b)). We vary the CPU budget allocated to rule processing (10%, 20%, 40%) in combination with zero, 4K or 6K memory at switches. For example in case "40_0" (i.e., each server has 40% CPU budget, but there is no capacity at switches), SourcePlacement results in an infeasible solution, since the highest CPU usage is 56% for range IP allocation and 42% for random IP allocation. In contrast, vCRIB can find feasible solutions in all the cases except "10_0" case. When we have only 10% CPU budget at servers, vCRIB needs some memory space at the switches (e.g., 4K rules) to find a feasible solution. With a 20% CPU budget, vCRIB can find a feasible solution even without any switch capacity ("20_0"). With higher CPU budgets, or with additional switch memory, vCRIB's traffic overhead becomes negligible. Thus, vCRIB can effectively manage heterogeneous resource constraints and find low traffic-overhead placement in these settings. Unlike with memory constraints, Range IP assignment with CPU constraints does not have a lower average load on servers for SourcePlacement, nor does it have a feasible solution with lower traffic overhead, since with the CPU resource usage function closer partitions in the source IP dimension are no longer the most similar.

4.3 Resource Usage and Traffic Spatial Distribution

We now study how resource usage and traffic overhead are spatially distributed across a data center for the Random case.



(a) Traffic overhead for different rules (b) Traffic overhead on different links (c) Memory usage on different devices

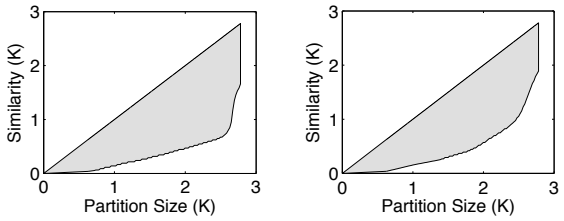
Figure 8: *Spatial distribution of traffic and resource usage*

vCRIB is effective in leveraging on-path and nearby devices. Figure 8(a) shows the case where servers have a capacity of 4K and switches have none. We classify the rules into deny rules, accept rules whose traffic stays within the rack (labelled as "ToR"), within the Pod ("Pod"), or goes through the core routers ("Core"). In general, vCRIB may redirect traffic to other locations away from the original paths, causing traffic overhead. We thus classify the traffic overhead based on the hops the traffic incurs, and then normalize the overhead based on the traffic volume in the SourcePlacement approach. Adding the percentage of traffic that is handled in the same rack of the source for deny traffic (8.8%) and source or destination for accept traffic (1.8% ToR, 2.2% POD, and 1.6% Core), shows that out of 21% traffic overhead, about 14.4% is handled in nearby servers.

Most traffic overhead vCRIB introduces is within the rack. Figure 8(b) classifies the locations of the extra traffic vCRIB introduces. vCRIB does not require additional bandwidth resources at the core links; this is advantageous, since core links can limit bisection bandwidth. In part, this can be explained by the fact that only 20% of our traffic traverses core links. However, it can also be explained by the fact that vCRIB places partitions only on ToRs or servers close to the source or destination. For example, in the "4K_0" case, there is 29% traffic overhead in the rack, 11% in the Pod and 2% in the core routers, and based on Figure 8(c) all partitions are saved on servers. However, if we add 4K capacity to internal switches, vCRIB will offload some partitions to switches close to the traffic path to lower the traffic overhead. In this case, for accept rules, the ToR switch is on the path of traffic and does not increase traffic overhead. Note that the servers are always full as they are the best place for saving partitions.

4.4 Parameter Sensitivity Analysis

The IP assignment method, traffic locality and rules in partitions can affect vCRIB performance in finding a feasible solution with low traffic. Our previous evaluations have explored *uniform* IP assignment for two extreme cases Range and Random above. We have also evaluated a skewed distribution of the number of IPs/VMs per machine but have not seen major changes in the traffic overhead. In this case, vCRIB was still able to find a nearby machine with lower load. We also conducted another experiment with different traffic locality patterns, which showed that having more non-local flows gives vCRIB more choices to offload rule processing and reach feasible solutions with lower traffic overhead. Finally, experiments on FFDS performance for different machine capacities [27] also validates its superior performance comparing to the tree-based placement [33]. Beyond these kinds of analyses, we have also explored the parameter space of similarity and partition size, which we discuss next.



(a) Feasibility region (b) 10% traffic overhead

Figure 9: *vCRIB working region and ruleset properties*

vCRIB uses similarity to accommodate larger partitions. We have explored two properties of the rules in partitions by changing the ruleset. In Figure 9, we define a two dimensional space: one dimension measures the average similarity between partitions and the other the average size of partitions. Intuitively, the size of partitions is a measure of the difficulty in finding a feasible solution and similarity is the property of a ruleset that vCRIB exploits to find solutions. To generate this figure, we start from an infeasible setting for SourcePlacement with a maximum of 5.7K rules for "4k_0" setting and then change the ruleset without changing the load on the maximum loaded server. We then explore the two dimensions as follows. Starting from the ClassBench ruleset and Range IP assignment, we split rules into half in the source IP dimension to decrease similarity without changing partition sizes. To increase similarity, we extend a rule in source IP dimension and remove rules in the extended area to maintain the same partition size.

Adding or removing rules matching only one VM (micro rules), also help us change average partitions size without changing the similarity. Unfortunately, removing just micro rules is not enough to explore the entire range of partition sizes, so we also remove rules randomly.

Figure 9(a) presents the *feasibility region* for vCRIB regardless of traffic overhead. Since average similarity cannot be more than the average partition size, the interesting part of the space is below the 45°. Note that vCRIB is able to cover a large part of the space. Moreover, the shape of the feasibility region shows that for a fixed average partition size, vCRIB works better for partitions with larger similarity. This means that to handle larger partitions, vCRIB needs more similarity between partitions; however, this relation is not linear since vCRIB may not be able to utilize the available similarity given limits on server capacity. When considering only solutions with less than 10% traffic overhead, vCRIB's feasibility region (Figure 9(b)) is only slightly smaller. This figure demonstrates vCRIB's utility: for a small additional traffic overhead, vCRIB can find many additional operating points in a data center that, in many cases, might have otherwise been infeasible.

We also tried a different method for exploring the space, by tuning the IP selection method on a fixed ruleset, and obtained qualitatively similar results [27].

4.5 Reaction to Cloud Dynamics

Figure 10 compares benefit-greedy (with timeout 10 seconds) with overhead-greedy and a randomized algorithm[7] after a single VM migration for the 4K_0 case. Each point in Figure 10 shows a step in which one partition is moved, and the horizontal axis is time in log scale. At time A, we migrate a VM from its current server S_{old} to a new one S_{new}, but S_{new} does not have any space for the partition of the VM, P. As a result, P remains on S_{old} and the traffic overhead increases by $40MBps$. Both benefit-greedy and overhead-greedy move the partition P for the migrated VM to a server in the rack containing S_{new} at time B and reduce traffic by 20Mbps. At time B, benefit-greedy brings out two partitions from their current host S_{new} to free up the memory for P while imposing a little traffic overhead. At time C, benefit-greedy moves P to S_{new} and reduces traffic further by $15Mbps$. The entire process takes only 5 seconds. In contrast, the randomized algorithm takes 100 seconds to find the right partitions and thus is not useful with these dynamics.

We then run multiple VM migrations to study the average behavior of benefit-greedy with 5 and 10 seconds timeout. In each 20 seconds interval, we randomly pick a VM and move it to another random server. Our simulations last for 30 minutes. The trend of data center traffic in Figure 11 shows that benefit-greedy maintains traffic levels, while overhead-greedy is unable to do so. Over time, benefit-greedy (both configurations) reduces the average traffic overhead around 34 MBps, while overhead-greedy algorithm increases the overhead by 117.3 MBps. Besides, this difference increases as the interval between two VM migration increases.

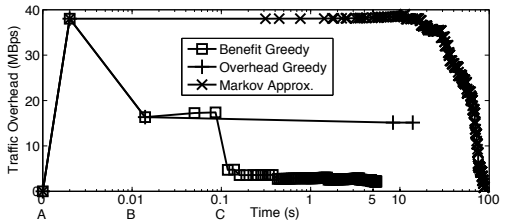


Figure 10: *Traffic refinement for one VM migration*

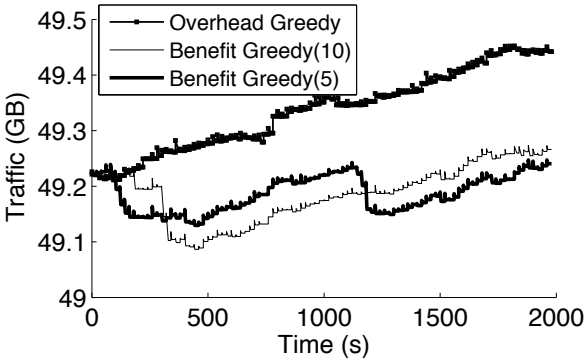


Figure 11: *The trend of traffic during multiple VM migration*

4.6 Prototype Evaluation

We built vCRIB prototype using Open vSwitch [4] as servers and switches, and POX [1] as the platform for vCRIB controller for micro-benchmarking.

Overhead of collecting traffic information: In our prototype, we send traffic information collected from each server's Open vSwitch kernel module to the controller. Each piece of information requires 13 Bytes for 5 tuples[8] and 2 Bytes for the traffic change volume.

Since we only need to detect traffic changes at the rule-level, we can more aggressively filter the traffic information than traditional traffic engineering solutions [11]. The vCRIB controller sets a threshold $\delta(F)$ for traffic changes of a set of flows F and sends the threshold to the servers. The servers then only report traffic changes above $\delta(F)$. We set the threshold δ for two different granularities of flow sets F. A larger set F makes vCRIB less sensitive to individual flow changes and leads to less reporting overhead but incurs less accuracy. (1) We set F as the volume each *rule* for each *destination server* in

[7] Markov Approximation [20] with target switch selection probability $\propto$ exp(traffic reduction of migration step)

[8] Some rules may have more packet header fields and thus require more bytes. In this cases, we can compress these information using fingerprints to reduce the overhead.

each *per-source partition*. (2) We assume all the rules in a partition have accept actions (as the worst case for traffic). Thus, the vCRIB controller sets the threshold that affects the size of traffic to each *destination server* for each *per-source partition* (summing up all the rules). If there are 20 flow changes above the threshold, we need to send 260B/s per server, which means 20Mbps for 10K servers in the data center. For VM migrations and rule insertion/deletion, the vCRIB controller can be notified directly by the the data center management system.

Controller overhead: We measure the delay of processing 200K ClassBench rules. Initially, the vCRIB controller partitions these rules, runs the resource-aware placement algorithm and the traffic-aware refinement to derive an initial placement; this takes up to *five* minutes. However, these recomputations are triggered only when a placement becomes infeasible; this can happen after a long sequence of rule changes or VM add/remove.

The traffic overhead of rule installation and removal depends on the number of refinement steps and the number of rules per partition. The size of OpenFlow command for a rule entry is 100 Bytes, so if a partition has 1K rules, the overhead of removing it from one device and installing at another device is 200KB. For each VM migration, which needs an average of 11 partitions, the bandwidth overhead of moving the rules is 11×200KB=2.2MB.

Reaction to cloud dynamics: We evaluate the latency of handling traffic changes by deploying our prototype in a topology with five switches and six servers as shown in Figure 1. We deploy a vCRIB controller that connects with all the devices with an RTT of 20 ms. We set the capacity of each server/switch as large enough to store at most one partition. We then inject a traffic change pattern that causes vCRIB to swap two partitions and add a redirection rule at a VM. It takes vCRIB *30ms* to detect the traffic changes, and move the rules to the new locations.

5 Related Work

Our work is inspired by several different strands of research, each of which we cover briefly.

Policies and rules in the cloud: Recent proposals for new policies often propose customized systems to manage rules on either hypervisors [4, 13, 32, 30]) or switches [3, 8, 29]. vCRIB proposes an abstraction of a centralized rule repository for all the policies, frees these systems from the complexity inherent in the rule management, and handles heterogeneous resource constraints at devices while minimizing the traffic overhead.

Rule management in software-defined networks (SDNs): Recent work on SDNs provides rule repository abstractions and some rule management capabilities [12, 23, 38, 13]. vCRIB focuses on data centers, which are more dynamic, more sensitive to traffic overhead, and face heterogeneous resource constraints.

Distributed firewall: Distributed firewalls [9, 19], often used in enterprises, leverage a centralized manager to deploy security policies on edge machines. vCRIB manages more fine-grained rules on flows and VMs for various policies including firewalls in the cloud. Rather than placing these rules at the edge, vCRIB places these rules taking into account the rule processing constraints, while minimizing traffic overhead.

Rule partition and placement solutions: The problem of partitioning and placing multi-dimensional data at different locations also appears in other contexts. Unlike traditional partitioning algorithms [36, 34, 16, 25, 24] which divide rules into partitions using a top-down approach, vCRIB uses *per-source partitions* to place the partitions close to the source with low traffic overhead. Compared with DIFANE [38], which *randomly* places *a single* partition of rules at each switch, vCRIB takes the *partitions-with-replication* approach to flexibly place *multiple* per-source partitions at one device. In preliminary work [26], we proposed an *offline* placement solution which works *only* for the TCAM resource model. The paper has a top-down heuristic partition-with-split algorithm which cannot limit the overhead of redirection rules and is not optimized for CPU-based resource model. Besides, having partitions with traffic from multiple sources requires complicated partition replication to minimize traffic overhead. In contrast, vCRIB uses fast per-source partition-with-replication algorithm which reduces TCAM-usage by leveraging similarity of partitions and restricts the resource usage of redirection by using limited number of equal shaped redirection rules. Our preliminary work used an unscalable DFS branch-and-bound approach to find a feasible solution and optimized the traffic in one step. vCRIB scales better using a two-phase solution where the first phase has an approximation bound in finding a feasible solution and the second can be run separately when the placement is still feasible.

6 Conclusion

vCRIB, is a system for automatically managing the fine-grained rules for various management policies in data centers. It jointly optimizes resource usage at both switches and hypervisors while minimizing traffic overhead and quickly adapts to cloud dynamics such as traffic changes and VM migrations. We have validated its design using simulations for large ClassBench rulesets and evaluation on a vCRIB prototype built on Open vSwitch. Our results show that vCRIB can find feasible placements in most cases with very low additional traffic overhead, and its algorithms react quickly to dynamics.

References

[1] http://www.noxrepo.org/pox/about-pox.

[2] http://www.praxicom.com/2008/04/the-amazon-ec2.html.

[3] Big Switch Networks. http://www.bigswitch.com/.

[4] Open vSwitch. http://openvswitch.org/.

[5] Private conversation with Amazon.

[6] Private conversation with rackspace operators.

[7] Virtual networking technologies at the server-network edge. http://h20000.www2.hp.com/bc/docs/support/SupportManual/c02044591/c02044591.pdf.

[8] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[9] S. M. Bellovin. Distributed Firewalls. *;login:*, November 1999.

[10] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, 2010.

[11] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *ACM CoNEXT*, 2011.

[12] M. Casado, M. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker. Rethinking Enterprise Network Control. *IEEE/ACM Transactions on Networking*, 17(4), 2009.

[13] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the Network Forwarding Plane. In *PRESTO*, 2010.

[14] C. Chekuri and S. Khanna. A PTAS for the Multiple Knapsack Problem. In *SODA*, 2001.

[15] A. Curtis, J. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling Flow Management for High-Performance Networks. In *SIGCOMM*, 2011.

[16] P. Gupta and N. McKeown. Packet Classification using Hierarchical Intelligent Cuttings. In *Hot Interconnects VII*, 1999.

[17] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Bannerjee, and N. McKeown. ElasticTree: Saving Energy in Data Center Networks. In *NSDI*, 2010.

[18] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical Declarative Network Management. In *WREN*, 2009.

[19] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a Distributed Firewall. In *CCS*, 2000.

[20] J. Jiang, T. Lan, S. Ha, M. Chen, and M. Chiang. Joint VM Placement and Routing for Data Center Traffic Engineering. In *INFOCOM*, 2012.

[21] E. G. C. Jr., M. R. Carey, and D. S. Johnson. Approximation Algorithms for NP-hard Problems. chapter Approximation Algorithms for Bin Packing: A Survey. PWS Publishing Co., Boston, MA, USA, 1997.

[22] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The Nature of Datacenter Traffic: Measurements and Analysis. In *IMC*, 2009.

[23] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.

[24] V. Kriakov, A. Delis, and G. Kollios. Management of Highly Dynamic Multidimensional Data in a Cluster of Workstations. *Advances in Database Technology-EDBT*, 2004.

[25] A. Mondal, M. Kitsuregawa, B. C. Ooi, and K. L. Tan. R-tree-based Data Migration and Self-Tuning Strategies in Shared-Nothing Spatial Databases. In *GIS*, 2001.

[26] M. Moshref, M. Yu, A. Sharma, and R. Govindan. vCRIB: Virtualized Rule Management in the Cloud. In *HotCloud*, 2012.

[27] M. Moshref, M. Yu, A. Sharma, and R. Govindan. vCRIB: Virtualized Rule Management in the Cloud. Technical Report 12-930, Computer Science, USC, 2012. http://www.cs.usc.edu/assets/004/83467.pdf.

[28] J. Mudigonda, P. Yalagandula, J. Mogul, and B. Stiekes. NetLord: A Scalable Multi-Tenant Network Architecture for Virtualized Datacenters. In *SIGCOMM*, 2011.

[29] L. Popa, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing The Network In Cloud Computing. In *HotNets*, 2011.

[30] L. Popa, M. Yu, S. Y. Ko, I. Stoica, and S. Ratnasamy. CloudPolice: Taking Access Control out of the Network. In *HotNets*, 2010.

[31] S. Rupley. Eyeing the Cloud, VMware Looks to Double Down On Virtualization Efficiency, 2010. `http://gigaom.com/2010/01/27/eyeing-the-cloud-vmware-looks-to-double-down-on-virtualization-efficiency`.

[32] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the Datacenter Networks. In *NSDI*, 2011.

[33] M. Sindelar, R. K. Sitaram, and P. Shenoy. Sharing-Aware Algorithms for Virtual Machine Colocation. In *SPAA*, 2011.

[34] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet Classification Using Multidimensional Cutting. In *SIGCOMM*, 2003.

[35] D. E. Taylor and J. S. Turner. ClassBench: A Packet Classification Benchmark. *IEEE/ACM Transactions on Networking*, 15(3), 2007.

[36] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. Efficuts: Optimizing Packet Classification for Memory and Throughput. In *SIGCOMM*, 2010.

[37] A. Voellmy, H. Kim, and N. Feamster. Procera: A Language for High-Level Reactive Network Control. In *HotSDN*, 2010.

[38] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-Based Networking with DIFANE. In *SIGCOMM*, 2010.

Chatty Tenants and the Cloud Network Sharing Problem

Hitesh Ballani[†] Keon Jang[†] Thomas Karagiannis[†]
Changhoon Kim[‡] Dinan Gunawardena[†] Greg O'Shea[†]

[†]*Microsoft Research* [‡]*Windows Azure*
Cambridge, UK *USA*

Abstract

The emerging ecosystem of cloud applications leads to significant inter-tenant communication across a datacenter's internal network. This poses new challenges for cloud network sharing. Richer inter-tenant traffic patterns make it hard to offer minimum bandwidth guarantees to tenants. Further, for communication between economically distinct entities, it is not clear whose payment should dictate the network allocation.

Motivated by this, we study how a cloud network that carries both intra- and inter-tenant traffic should be shared. We argue for network allocations to be dictated by the least-paying of communication partners. This, when combined with careful VM placement, achieves the complementary goals of providing tenants with minimum bandwidth guarantees while bounding their maximum network impact. Through a prototype deployment and large-scale simulations, we show that minimum bandwidth guarantees, apart from helping tenants achieve predictable performance, also improve overall datacenter throughput. Further, bounding a tenant's maximum impact mitigates malicious behavior.

1 Introduction

As cloud platforms mature, applications running in cloud datacenters increasingly use other cloud-based applications and services. Some of these services are offered by the cloud provider; for instance, Amazon EC2 offers services like S3, EBS, DynamoDB, RDS, SQS, CloudSearch, etc. that tenants can use as application building blocks [1]. Other services like CloudArray and Alfresco are run by tenants themselves [2]. The resulting ecosystem of applications and services means that, apart from communication between virtual machines of the same tenant, there is an increasing amount of tenant-tenant and tenant-provider communication [3]. Indeed, examining several datacenters of a major cloud provider, we find that such inter-tenant traffic can amount up to 35% of the total datacenter traffic. Looking ahead, we expect this ecosystem to become richer, further diversifying network traffic in the cloud.

The increasing importance and diversity of network traffic in cloud datacenters is at odds with today's cloud platforms. While tenants can rent virtual machines (VMs) with dedicated cores and memory, the underlying network is shared. Consequently, tenants experience variable and unpredictable network performance [4–6] which, in turn, impacts both application performance and tenant costs [5,7,8]. To tackle this, many network sharing policies have been proposed [9–14]. In recent work, FairCloud [14] presented a set of requirements for network sharing: i). associate VMs with minimum bandwidth guarantees, ii). ensure high network utilization, and iii). divide network resources in proportion to tenant payments. However, the proposals above focus only on intra-tenant communication, and naively extending these requirements to inter-tenant settings is problematic.

Inter-tenant traffic changes the network sharing problem both quantitatively and qualitatively, and leads to two main challenges. First, offering (non-trivial) minimum bandwidth guarantees for inter-tenant traffic is harder as the set of VMs that can possibly communicate with each other is significantly larger. Second, traffic flowing between distinct economic entities begs the question– whose payment should the bandwidth allocation be proportional to? Extending intra-tenant proportionality [14] to inter-tenant scenarios entails that bandwidth should be allocated in proportion to the combined payment of communicating partners. However, this is inadequate as tenants can increase their network allocation, beyond what their payment dictates, by communicating with more VMs of other tenants. Thus, the challenge is how to achieve proportional yet robust network sharing in inter-tenant scenarios.

Motivated by these challenges, we revisit the requirements for sharing a cloud network. To address the first

challenge of providing minimum bandwidth guarantees, we propose relaxing the semantics of the guarantees offered. For example, instead of targeting arbitrary communication patterns, a VM is only guaranteed bandwidth for intra-tenant traffic and for traffic to tenants it depends upon. Such *communication dependencies* could be explicitly declared or inferred. To address the second challenge, we identify a new network sharing requirement, *upper-bound proportionality*, which requires that the maximum bandwidth a tenant can acquire is in proportion to its payment. This mitigates aggressive tenant behavior and ensures robust network sharing. We show this requirement can be met by allocating bandwidth to flows in proportion to the *least-paying* communication partner. We call this "Hose-compliant" allocation.

To illustrate these ideas, we design *Hadrian*, a network sharing framework for multi-tenant datacenters. With Hadrian, VMs are associated with a minimum bandwidth. This minimum guarantee and tenant dependencies guide the placement of VMs across the datacenter. Network bandwidth is allocated using the hose-compliant allocation policy. This, when combined with careful VM placement, achieves the complementary goals of providing tenants with minimum bandwidth while achieving upper-bound proportionality.

As a proof of concept, we have implemented a Hadrian prototype comprising an end-host and a switch component. Through testbed deployment and cross-validated simulation experiments, we show that Hadrian benefits both tenants and providers. Minimum VM bandwidth yields predictable and better network performance for tenants (at the 95^{th} percentile, flows finish 3.6x faster). For the provider, such guarantees improve datacenter throughput up to 20% by avoiding outliers with very poor network performance. Thus, providers can offer an improved service at a lower price while remaining revenue neutral.

Overall, our main contributions are–

- We provide evidence of the prevalence of inter-tenant traffic through measurements from eight datacenters of a major public cloud provider.
- We present a new definition of payment proportionality that ensures robust network sharing in inter-tenant scenarios. We also devise a bandwidth allocation policy that meets this proportionality.
- We present relaxed bandwidth guarantee semantics to improve the multiplexing a provider is able to achieve. Further, we design a novel VM placement algorithm that uses a max-flow network formulation to satisfy such guarantees.
- To illustrate the feasibility of the mechanisms above, we present the design and implementation of Hadrian.

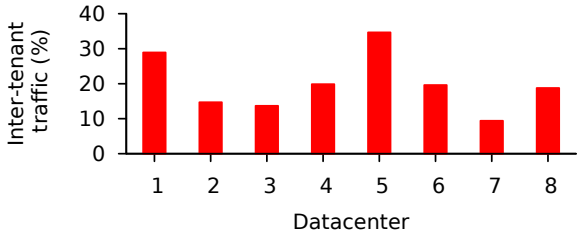


Figure 1: Inter-tenant traffic, as a % of the datacenter's total traffic, for eight production datacenters.

2 Motivation and challenges

With Infrastructure-as-a-Service, tenants rent VMs with varying amount of CPU, memory and storage, and pay a fixed flat rate per-hour for each VM [15,16]. However, the cloud's internal network is shared amongst the tenants. The cloud network carries *external* traffic to and from the Internet, *intra-tenant* traffic between a tenant's VMs and *inter-tenant* traffic. The latter includes traffic between different customer tenants, and between customer tenants and provider services. In the rest of this paper, we use the term "tenant" to refer to both customer tenants and provider services.

In this section, we show the prevalence of inter-tenant traffic in datacenters, and use this to drive our design.

2.1 Inter-tenant communication

Today's cloud providers offer many services that tenants can use to compose their cloud applications. For example, Amazon AWS offers sixteen services that result in network traffic between tenant VMs and service instances [1]. These services provide diverse functionality, ranging from storage to load-balancing and monitoring. Over a hundred cloud applications using such services are listed here [17]. Beyond provider services, many tenants run cloud applications that provide services to other tenants. AWS marketplace [2] lists many such tenant services, ranging from web analytics to identity and content management. These result in tenant-tenant traffic too.

Chatty tenants. To quantify the prevalence of inter-tenant traffic in today's datacenters, we analyze aggregate traffic data collected from eight geographically distributed datacenters operated by a major public cloud provider. Each datacenter has thousands of physical servers and tens of thousands of customer VMs. The data was collected from August 1-7, 2012 and includes all traffic between customer VMs. This does not include traffic to and from provider services. The total volume of such traffic in each datacenter was a few hundred terabytes to a few petabytes. Figure 1 shows that the percentage of inter-tenant traffic varies from 9 to 35%. When external traffic to or from the Internet is excluded, inter-tenant traffic varies from 10 to 40%.

We also studied traffic between customer tenants and

a specific provider service, the storage service. Since the local disk on a VM only provides ephemeral storage, for persistent storage, all tenants rely on the storage service which needs to be accessed across the network. The resulting network traffic is inter-tenant too. By analyzing application-level logs of the storage service, we found that read and write traffic for the storage service is, on average, equal to 36% and 70% of the total traffic between customer VMs respectively.

We complement these public cloud statistics with data from a private cloud with ∼300 servers. This represents a mid-size enterprise datacenter. The servers run over a hundred applications and production services, most serving external users. In summary, we find that 20% of the servers are involved in inter-tenant communication. For these servers, the fraction of inter-tenant to total flows is 6% at the median and 20% at the 95^{th} percentile with a maximum of 56% (3% at the median and 10% at the 95^{th} percentile in terms of volume). A tenant communicates with two other tenants in the median case and six at the 95^{th} percentile. Further, this inter-tenant traffic is not sporadic as it is present even at fine timescales.

Overall we find that tenants are indeed chatty with a significant fraction of traffic between tenants.

Impact of network performance variability. Several studies have commented on variable network performance in datacenters [4–6]. This impacts both provider and tenant services. For example, the performance of the cloud storage service varies both over time and across datacenters [18,19]. Any of the resources involved can cause this: processing on VMs, processing or disks at the storage tier, or the cloud network. For large reads and writes, the network is often the bottleneck resource. For example, Ghosal et al. [18] observed a performance variability of >2x when accessing Amazon EBS from large VMs (and far greater variability for small VMs), a lot of which can be attributed to cloud network contention.

In summary, these findings highlight the significance of inter-tenant traffic in today's datacenters. As cloud service marketplaces grow, we expect an even richer ecosystem of inter-tenant relationships. Offering service-level agreements in such settings requires network guarantees for inter-tenant communication.

2.2 Cloud network sharing

The distributed nature of the cloud network makes it tricky to apportion network bandwidth fairly amongst tenants. Today, network bandwidth is allocated through end host mechanisms such as TCP congestion control which ensure per-connection fairness. This has a number of drawbacks. For instance, misbehaving tenants can unfairly improve their network performance by using multiple TCP connections [12] or simply using UDP. Consequently, the cloud network sharing problem has received a lot of attention [9–14]. This section describes how inter-tenant traffic adds a new dimension to this problem.

2.2.1 Network sharing requirements

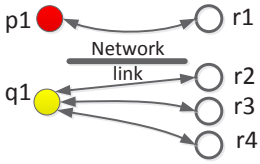
We first discuss how a cloud network carrying only intra-tenant traffic should be shared. FairCloud [14] presented the following broad set of requirements for fairly sharing the cloud network with a focus on intra-tenant traffic.
(1). **Min-Guarantee**: Each VM should be guaranteed a minimum bandwidth. This allows tenants to estimate worst-case performance and cost for their applications.
(2). **High Utilization**: Cloud datacenters multiplex physical resources across tenants to amortize costs and the same should hold for the network. So, spare network resources should be allocated to tenants with demand (work conservation). Further, tenants should be incentivised to use spare resources.
(3). **Proportionality**: Just like CPU and memory, the network bandwidth allocated to a tenant should be proportional to its payment.

However, inter-tenant traffic has important implications for these sharing requirements. As explained below, such traffic makes it harder to offer minimum bandwidth guarantees and necessitates a different kind of proportionality. Overall, *we embrace the first and second requirements, and propose a new proportionality requirement suitable for inter-tenant settings*.

2.2.2 Implications of inter-tenant traffic

Min-guarantee. Guaranteeing the minimum bandwidth for a VM requires ensuring sufficient capacity on all network links the VM's traffic can traverse. For intra-tenant traffic, this is the set of network links connecting a tenant's VMs. However, for inter-tenant traffic, the set expands to network links between VMs of all tenants that may communicate with each other. If we assume no information about a tenant's communication partners, the minimum bandwidth for each VM needs to be carved on all network links, and is thus strictly limited by the capacity of the underlying physical network. For instance, consider a datacenter with a typical three-tier tree topology with a 1:4 oversubscription at each tier (i.e., core links are oversubscribed by 1:64). If each physical server has 4 VMs and 1 Gbps NICs, such naive provisioning would provide each VM with a minimum guarantee of a mere 4 Mbps (≈ 1000/(4*64))! Hence, *richer traffic patterns resulting from inter-tenant communication make it harder to guarantee minimum bandwidth for VMs*.

Payment proportionality. Defining payment proportionality for inter-tenant settings is tricky. Since traffic can flow between different tenants, a key question is whose payment should dictate the bandwidth it gets. Intra-tenant proportionality requires that a tenant be allocated bandwidth in proportion to its payment. A simple



(a)

Allocation	P	Q	R
per-flow	250	750	1000
per-src (→)	500	500	1000
per-src (←)	250	750	1000
PS-L	333	666	1000
PS-P	250	750	1000

(b)

Figure 2: Inter-tenant traffic: Tenants P, Q and R have one ($p1$), one ($q1$) and four VMs respectively.

extension to inter-tenant settings entails that if a set of tenants are communicating with each other, their combined bandwidth allocation should be proportional to their combined payment. Assuming tenants pay a fixed uniform price for each VM, this means that a tenant's bandwidth should be in proportion to the total number of VMs involved in its communication (its own VMs and VMs of other tenants too).

As an example, consider the inter-tenant scenario shown in Figure 2. Since the traffic for tenants P and Q involves 2 and 4 VMs respectively, such proportionality requires that tenant Q's bandwidth be twice that of P, i.e., $\frac{B_Q}{B_P} = \frac{4}{2}$. Similarly, $\frac{B_R}{B_P} = \frac{6}{2}$ and $\frac{B_R}{B_Q} = \frac{6}{4}$. Further, the high utilization requirement entails that the link's capacity (1000 Mbps) should be fully utilized, i.e., $B_P + B_Q = B_R = 1000$. An allocation where $B_P = 333$, $B_Q = 666$ and $B_R = 1000$ satisfies these requirements.

While past proposals for network sharing have all focused on intra-tenant traffic, we consider how they would fare in this inter-tenant scenario. Figure 2b shows the bandwidth for tenants P, Q and R with different allocation strategies. Per-flow allocation ensures each flow gets an equal rate. Here, "flow" refers to the set of connections between a given pair of VMs. Hence tenant P, with its one flow, gets a quarter of the link's capacity, i.e., $B_P = 250$. Per-source allocation [12] gives an equal rate to each source, so the bandwidth allocated depends on the direction of traffic. PS-L and PS-P are allocation strategies that assign bandwidth in a weighted fashion with carefully devised weights for individual flows [14]. As the table shows, only PS-L, which was designed to achieve intra-tenant proportionality, satisfies the extended proportionality definition too.

However, *such proportionality means that a tenant can acquire a disproportionate network share by communicating with more VMs of other tenants*. For example, in Figure 2b, all approaches result in a higher bandwidth for tenant Q than P because Q is communicating with more VMs, even though both P and Q pay for a single VM and are using the same service R. Q could be doing this maliciously or just because its workload involves more communication partners. Further, Q can increase its share of the link's bandwidth by communicating with even more VMs. This key property leads to two problems. First, this makes it infeasible to guarantee a minimum bandwidth for any VM. In effect, such proportionality is incompatible with the min-guarantee requirement. Second, it allows for network abuse. An aggressive tenant with even a single VM can, simply by generating traffic to VMs of other tenants, degrade the performance for any tenants using common network links. The presence of many cloud services that are open to all tenants makes this a real possibility.

The root cause for these problems is that, with current network sharing approaches in inter-tenant settings, there is no limit on the fraction of a link's bandwidth a VM can legitimately acquire. To avoid such unbounded impact, we propose a new network sharing requirement.

Requirement 3. Upper bound proportionality: The maximum bandwidth each tenant and each VM can acquire should be a function of their payment. Further, this upper bound should be independent of the VM's communication patterns. Later we show that, apart from mitigating tenant misbehavior, *this new proportionality definition is compatible with the min-guarantee requirement*. It actually facilitates offering minimum VM bandwidth.

3 Revisiting network sharing

Guided by the observations above, we study how a cloud network that carries both intra- and inter-tenant traffic should be shared. The sharing should meet three requirements: (i). Minimum bandwidth guarantees, (ii). High Utilization, and (iii). Upper bound proportionality.

3.1 Minimum bandwidth guarantee

The cloud provider may allow tenants to specify the minimum bandwidth for individual VMs. So each VM p is associated with a minimum bandwidth B_p^{min}. Alternatively, the provider may offer a set of VM classes with varying guarantees (small, medium, and large, as is done for other resources today). In either case, a VM's minimum guarantee should dictate its price.

Like past proposals [10,11,14,20,21], we use the hose model to capture the semantics of the bandwidth guarantees being offered. As shown in Figure 3, with this model a tenant can imagine each of its VMs is connected to an imaginary, non-blocking switch by a link whose capacity is equal to the VM's minimum bandwidth. For simplicity, we assume all VMs for a tenant have the same minimum bandwidth.

However, as described in §2.2.2, richer inter-tenant communication patterns severely limit that the provider's ability to accommodate many concurrent tenants with minimum bandwidth guarantees atop today's oversubscribed networks. We show this in our experiments too. To better balance the competing needs of tenants and providers, we propose relaxing the bandwidth guarantees offered to tenants; they should be reasonable for tenants yet provider friendly. To achieve this, we rely on: i).

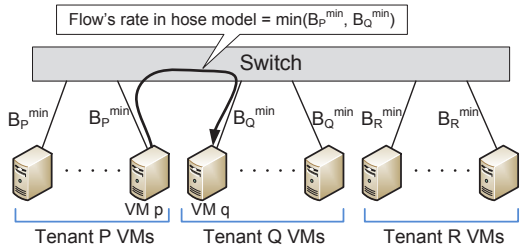


Figure 3: Hose model with three tenants.

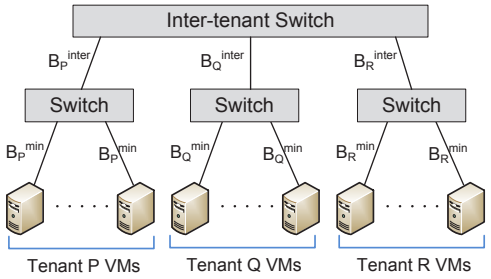


Figure 4: Hierarchical hose model gives per-VM minimum bandwidth for intra-tenant traffic and per-tenant minimum for inter-tenant traffic.

communication dependencies, and ii). hierarchical guarantees. We elaborate on these below.

3.1.1 Communication dependencies

Allowing arbitrary VMs to communicate under guaranteed performance is impractical. Instead, our guarantees apply only for "expected" communication. To this end, we rely on communication dependencies. A tenant's communication dependency is a list of other tenants or peers that the tenant expects to communicate with. Examples of such dependencies include: i) P: {Q}, ii) Q: {P, R}, iii) R: {*}.

The first dependency is declared by tenant P and implies that VMs of P, apart from sending traffic to each other, should be able to communicate with VMs of tenant Q. The second dependency is for Q and declares its peering with tenants P and R. Since a tenant running a service may not know its peers a priori, we allow for wildcard dependencies. Thus, the last dependency implies that tenant R is a service tenant and can communicate with any tenant that explicitly declares a peering with R (in this example, tenant Q). Note however that since P has not declared a peering with R, communication between VMs of P and R is not allowed.

The provider can use these dependencies to determine what inter-tenant communication is allowed and is thus better positioned to offer bandwidth guarantees to tenants. In the example above, the communication allowed is $P \leftrightarrow Q$ and $Q \leftrightarrow R$. An additional benefit is that this makes the cloud network "default-off" [22] since traffic can only flow between a pair of tenants if both have declared a peering with each other. This is in contrast to the "default-on" nature of today's cloud network.

We admit that discovering communication dependencies is challenging. While tenants could be expected to declare their dependencies when asking for VMs, a better option may be to infer them automatically. For example, today tenants need to sign up for provider services, so such tenant-provider dependencies are known trivially. The same mechanism could be extended for third-party services.

3.1.2 Hierarchical guarantees

With the hose model, each VM gets a minimum guarantee for all its traffic, irrespective of whether the traffic is destined to the same tenant or not. However, when accessing other tenants and services, tenants may find it easier to reason about an aggregate guarantee for all their VMs. Further, typical cloud applications involve more communication between VMs of the same tenant than across tenants. This was observed in our traces too. To account for these, we introduce hierarchical guarantees. Figure 4 shows the hierarchical hose model that captures such guarantees. Each VM for tenant P is guaranteed a bandwidth no less than B_P^{min} for traffic to P's VMs. Beyond this, the tenant also gets a minimum bandwidth guarantee for its aggregate inter-tenant traffic, B_P^{inter}.

Putting these modifications together, we propose offering tenants hose-style guarantees combined with communication dependencies and hierarchy. Hence, a tenant requesting V VMs is characterized by the four tuple $<V, B^{min}, B^{inter}, dependencies>$.[1] We show in §5.1 how this allows the provider to achieve good multiplexing while still offering minimum bandwidth to tenants.

3.2 Upper bound proportionality

Upper bound proportionality seeks to tie the maximum bandwidth a tenant can acquire to its payment. The same applies for individual VMs. Since a VM's minimum bandwidth dictates its price, it can be used as a proxy for payment. Thus, such proportionality requires that the upper bound on the aggregate bandwidth allocated to a VM should be a function of its minimum bandwidth. In this section, we describe a bandwidth allocation policy that achieves such proportionality.

3.2.1 Hose-compliant bandwidth allocation

Allocating bandwidth to flows in proportion to the combined payment of communicating partners results in tenants being able to get a disproportionate share of the network. To avoid this, we argue for bandwidth to be allocated in proportion to the least paying of communication partners. In other words, the bandwidth allocated to a flow should be limited by both source and destination

[1] Typically $B^{inter} < V * B^{min}$. If $B^{inter} = V * B^{min}$, no hierarchy is used and VMs simply get the same minimum bandwidth for all their traffic.

payment. For example, consider a flow between VMs p and q belonging to tenants P and Q. The minimum bandwidth for these VMs is B_P^{min} and B_Q^{min}, and they have a total of N_p and N_q flows respectively. Note that B_P^{min} reflects the payment for VM p. Assuming a VM's payment is distributed evenly amongst its flows, p's payment for the p–q flow is B_P^{min}/N_p. Similarly, q's payment for the flow is B_Q^{min}/N_q. Hence, this allocation policy says the flow should be allocated bandwidth in proportion to the smaller of these values, i.e., $\min(\frac{B_P^{min}}{N_p}, \frac{B_Q^{min}}{N_q})$.[2]

To achieve this, we assign appropriate weights to flows and allocate them network bandwidth based on weighted max-min fairness. So the bandwidth for a flow between VMs p and q, as determined by the bottleneck link along its path, is given by

$$B_{p,q} = \frac{w_{p,q}}{w_T} * C, \quad \text{where } w_{p,q} = min(\frac{B_P^{min}}{N_p}, \frac{B_Q^{min}}{N_q}) \quad (1)$$

Here, $w_{p,q}$ is the weight for this flow, C is the capacity of the bottleneck link and w_T is the sum of the weights for all flows across the link. Note that the weight for a flow is equal to the rate the flow would achieve on the hose model. Hence, we call this allocation "*Hose-compliant*".

Below we discuss how hose-compliance leads to upper bound proportionality and can also meet the other two network sharing requirements.

Req 3. Upper-bound Proportionality. Hose-compliant bandwidth allocation satisfies upper bound proportionality. The intuition here is that since the weight for each flow is limited by both the source and destination payments, the aggregate weight for a VM's flows and hence, its aggregate bandwidth has an upper bound. Formally, the aggregate weight for a VM p's flows–

$$w_p^{aggregate} = \sum_{q \in dst(p)} w_{p,q} = \sum_q min(\frac{B_P^{min}}{N_p}, \frac{B_Q^{min}}{N_q})$$

$$\Rightarrow w_p^{aggregate} \leq \sum_q \frac{B_P^{min}}{N_p} = \frac{B_P^{min}}{N_p} * N_p = B_P^{min} \quad (2)$$

So the aggregate weight for a VM's flows cannot exceed its minimum bandwidth. This aggregate weight, in turn, dictates the VM's aggregate bandwidth. This means that a tenant cannot acquire bandwidth disproportionate to its payment. More precisely, this yields the following constraint for a VM's total bandwidth on any link–

$$B_p = \sum_{q \in dst(p)} \frac{w_{p,q}}{w_T} * C \; = \; \frac{w_p^{aggregate}}{w_T} * C \; \leq \; \frac{B_P^{min}}{B_P^{min} + w_{T'}} * C$$

where B_p is the total bandwidth for VM p on the link, $w_{T'}$ is the sum of weights for all non-p flows and C is the link capacity. *Hence, hose-compliant allocation results in an upper bound for a VM's bandwidth on any link. This upper bound depends on the VM's minimum bandwidth (and hence, its payment).*

To understand this, let's revisit the inter-tenant scenario in Figure 2. Assume a minimum bandwidth of 100 Mbps for all VMs. With hose-compliant allocation, the weight for the $p1$–$r1$ flow is $\min(\frac{100}{1}, \frac{100}{1}) = 100$ while the weight for $q1$–$r2$ flow is $\min(\frac{100}{3}, \frac{100}{1}) = \frac{100}{3}$. Similarly, the weight for the $q1$–$r3$ and $q1$–$r4$ flow is $\frac{100}{3}$ too. Hence, the actual bandwidth for the $p1$–$r1$ flow is 500, while the other three flows get $\frac{500}{3}$ each. Note that even though tenant Q has three flows, their aggregate weight is the same as the weight for P's single flow. Hence, both tenants get the same bandwidth. This is desirable as both of them pay for a single VM. Even if tenant Q were to communicate with more VMs, the aggregate weight of its flows will never exceed 100. Thus, by bounding the aggregate weight for a VM's traffic, we ensure an upper bound for the impact it can have on any network link and hence, on the datacenter network.

Req 1. Min-guarantee. Minimum guarantees for VMs can be used to determine the minimum bandwidth that their flows should achieve. For example, in Figure 3, if VMs p and q communicate with N_p and N_q VMs each, then the bandwidth for a p–q flow should be at least $\min(\frac{B_P^{min}}{N_p}, \frac{B_Q^{min}}{N_q})$. With hose-compliant allocation, this is also the flow's weight $w_{p,q}$. This observation simplifies ensuring that flows do get their minimum bandwidth.

To ensure a flow's actual bandwidth always exceeds its guarantee, the total weight for all traffic that can traverse a link should not exceed its capacity. Formally, to ensure $B_{p,q} \geq w_{p,q}$, we need $w_T \leq C$ (see equation 1). This condition can be used to design a VM placement algorithm that ensures the condition holds across all links in the datacenter. §4.1 presents such an algorithm.

Req 2. High utilization. Hose-compliant allocation is work conserving. Since flows are assigned bandwidth in a weighted fashion, any VM with network demand is allowed to use spare capacity on network links. Beyond work conservation, high utilization also requires that tenants not be disincentivised to use spare bandwidth. This can be achieved by making the flow weights vary from link-to-link, as proposed in [14]. However, for brevity, we omit this extension in the rest of the paper.

4 Hadrian

Apart from a policy for bandwidth allocation, a complete network sharing solution has to include an admission control and VM placement mechanism to achieve proper network sharing. Guided by this, we design Hadrian, a

[2] A VM may favor some flows over others and choose to distribute its payment unevenly across its flows. This can be used by service providers to offer differentiated services to their clients. The allocation policy can accommodate such scenarios.

network sharing framework for multi-tenant datacenters that caters to both intra- and inter-tenant communication. Hadrian relies on the following two components.

• *VM Placement*. A logically centralized placement manager, upon receiving a tenant request, performs admission control and maps the request to datacenter servers. This allocation of VMs to physical servers accounts for the minimum bandwidth requirements and communication dependencies of tenants.

• *Bandwidth Allocation*. Hose-compliant allocation is used to assign network bandwidth to flows.

4.1 VM placement

VM placement problems are often mapped to multi-dimensional packing with constraints regarding various physical resources [23]. Our setting involves two resources– each tenant requires empty VM slots on physical servers and minimum bandwidth on the network links connecting them. *The key novelty in our approach is that we model minimum bandwidth requirements and communication dependencies of tenants as a max-flow network*. This allows us to convert our two-dimensional placement constraints into a simple set of constraints regarding the number of VMs that can be placed in a given part of the datacenter.

The placement discussion below focuses on tree-like physical network topologies like the multi-rooted tree topologies used today. Such topologies are hierarchical, made up of sub-trees at each level. Also, it assumes that if the topology offers multiple paths between VMs, the underlying routing protocol load balances traffic across them. This assumption holds for fat-tree topologies [24,25] that use multi-pathing mechanisms like ECMP, VLB [24] and Hedera [26].

4.1.1 Characterizing bandwidth requirements

Hose-compliant bandwidth allocation simplifies the problem of ensuring minimum VM bandwidth. As explained in §3.2.1, to satisfy the minimum guarantees of VMs, the provider needs to ensure the total weight for all traffic that can traverse any network link should not exceed the link's capacity. Thus, we need to quantify the total weight for traffic across any given link. To explain our approach, we use an example scenario involving three tenants, P, Q and R across any network link. The link has p VMs for tenant P to the left and the remaining p' VMs to the right. Similarly, there are q and r VMs for Q and R on the left, and q' and r' VMs on the right.

With hose-compliant bandwidth allocation, the aggregate weight for any VM's traffic cannot exceed its minimum guarantee (equation 2). So the total weight for traffic from all VMs on the left of the link cannot exceed the sum of their minimum bandwidths, i.e., $\sum(pB_P^{min} + qB_Q^{min} + rB_R^{min})$. The same holds for VMs on the right of the link. Further, since the weight for any given flow is limited by both its source and destination, the total weight for all traffic across the link is limited by both the total weight for VMs on the left and for VMs on the right of the link.

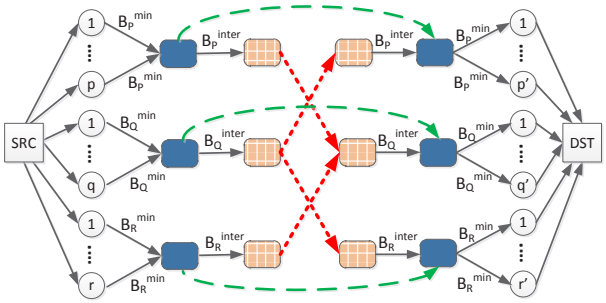


Figure 5: Flow network to capture the bandwidth needed on a link that connects p VMs of tenant P on the left to p' VMs on the right, and so on for tenants Q and R. Circles represent VMs, solid rectangles are intra-tenant nodes and shaded rectangles are inter-tenant nodes.

However, this analysis assumes all VMs can talk to each other and the same guarantees apply to both intra- and inter-tenant traffic. Accounting for communication dependencies and hierarchical guarantees leads to even more constraints regarding the total weight for the link's traffic. To combine these constraints, we express them as a **flow network**. A flow network is a directed graph where each edge has a capacity and can carry a flow not exceeding the capacity of the edge. Note that this flow is different from "real" flows across the datacenter network. Hereon, we use "link" to refer to physical network links while "edge" corresponds to the flow network.

Figure 5 shows the flow network for the link in our example and is explained below. All unlabeled edges have an infinite capacity. Each VM to the left of the physical link is represented by a node connected to the source node, while each VM to the right of the link is represented by a node connected to the destination. The VM nodes for any given tenant are connected to "intra-tenant" nodes (solid rectangles) by edges whose capacity is equal to the minimum bandwidth for the VM. These edges represent the constraint that the weight for a VM's traffic cannot exceed its minimum bandwidth. The two intra-tenant nodes for each tenant are connected by an edge of infinite capacity (long-dashed edge). Further, the two intra-tenant nodes for each tenant are connected to "inter-tenant" nodes (shaded rectangles) by edges whose capacity is equal to the tenant's minimum inter-tenant bandwidth. This constrains the total weight for inter-tenant traffic that each tenant can generate. Finally, based on the tenant communication dependencies, the appropriate inter-tenant nodes are connected to each

other (short-dashed edges). For our example, tenant Q can communicate with P and R, so inter-tenant nodes of Q are connected to those of P and R.

The max-flow for this flow network gives the total weight for all traffic across the link. This, in turn, is the *bandwidth required* on the physical link to ensure that the bandwidth guarantees of VMs are met.

4.1.2 Finding Valid Placements

Given a tenant request, a valid placement of its VMs should satisfy two constraints. First, VMs should only be placed on empty slots on physical hosts. Second, after the placement, the bandwidth required across each link in the datacenter should not exceed the link's capacity. The *VM Placement problem* thus involves finding a valid placement for a tenant's request. We designed a greedy, first-fit placement algorithm that we briefly sketch here. A formal problem definition, algorithm details and pseudo code are available in [27].

Instead of trying to place VMs while satisfying constraints across two dimensions (slots and bandwidth), we use the flow-network formulation to convert the bandwidth requirements on each physical link to constraints regarding the number of VMs that can be placed inside the sub-tree under the link, i.e., in the host, rack or pod under the link. Hence, given a tenant request, its placement proceeds as follows. We traverse the network topology in a depth-first fashion. Constraints for each level of the topology are used to recursively determine the maximum number of VMs that can be placed at sub-trees below the level, and so on till we determine the number of VMs that can be placed on any given host. VMs are then greedily placed on the first available host, and so on until all requested VMs are placed. The request is accepted only if all VMs can be placed.

A request can have many valid placements. Since datacenter topologies typically have less bandwidth towards the root than at the leaves, the optimization goal for the placement algorithm is to choose placements that reduce the bandwidth needed at higher levels of the datacenter hierarchy. To achieve this, we aim for *placement locality* which comprises two parts. First, a tenant's VMs are placed close to VMs of existing tenants that it has communication dependencies with. Second, the VMs are placed in the smallest sub-tree possible. This heuristic reduces the number and the height of network links that may carry the tenant's traffic. It preserves network bandwidth for future tenants, thus improving the provider's ability to accommodate them.

4.2 Bandwidth allocation

Hadrian uses hose-compliant bandwidth allocation. This can be achieved in various ways. At one end of the spectrum is an end-host only approach where a centralized controller monitors flow arrivals and departures to calculate the weight and hence, the rate for individual flows. These rates can then be enforced on physical servers. At the other end is a switch-only approach where switches know the VM bandwidth guarantees and use weighted fair queuing to achieve weighted network sharing. Both these approaches have drawbacks. The former is hard to scale since the controller needs to track the utilization of all links and all flow rates. The bursty nature of cloud traffic makes this particularly hard. The latter requires switches to implement per-flow fair queuing.

We adopt a hybrid approach involving end-hosts and switches. The goal here is to minimize the amount of network support required by moving functionality to trusted hypervisors at ends. Our design is based on explicit control protocols like RCP [28] and XCP [29] that share bandwidth equally and explicitly convey flow rates to end hosts. Hose-compliance requires weighted, instead of an equal, allocation of bandwidth. We provide a design sketch of our bandwidth allocation below.

To allow VMs to use any and all transport protocols, their traffic is tunneled inside hypervisor-to-hypervisor flows such that all traffic between a pair of VMs counts as one flow. Traffic is only allowed to other VMs of the same tenant and to VMs of peers. The main challenge is determining the rate for a flow which, in turn, is dictated by the flow's weight. The weight for a p–q flow is $\min(\frac{B_P^{min}}{N_p}, \frac{B_Q^{min}}{N_q})$. The source hypervisor hosting VM p knows B_P^{min} and N_p while the destination hypervisor knows B_Q^{min} and N_q. Thus, the source and destination hypervisor together have all the information to determine a flow's weight. For each flow, the hypervisor embeds B^{min} and N into the packet header, and over the course of the first round trip, the hypervisors at both ends have all the information to calculate the weights.

Packet headers also contain the flow weight. For the first round trip, hypervisors set the weight to a default value. Switches along the path only track the sum of the weights, S, for all flows through them. For a flow with weight w, its rate allocation on a link of capacity C is $\frac{w}{S} * C$. Each switch adds this rate allocation to the packet header. This reaches the destination and is piggybacked to the source on the reverse path. The source hypervisor enforces the rate it is allocated, which is the minimum of the rates given by switches along the path. To account for queuing or under-utilization due to insufficient demand, switches adjust C using the RCP control equation.

This basic design minimizes switch overhead; they do not need to maintain per-flow state. However, it does not support hierarchical guarantees. With such guarantees, the weight for a flow between different tenants depends on the total number of inter-tenant flows each of them have. Hence, hypervisors at the end of a flow do

not have enough information to determine its weight. Instead, switches themselves have to calculate the flow weight. In this extended design, the hypervisor also embeds the `VM-id`, the `tenant-id`, the inter-tenant bandwidth guarantee for the source and destination VM in the packet header. Each switch maintains a count of the number of inter-tenant flows for each tenant traversing it. Based on this, the switch can determine the weight for any flow and hence, its rate allocation. Thus switches maintain per-tenant state. The details for this extended design are available in [27].

4.3 Implementation

Our proof-of-concept Hadrian implementation comprises two parts.

(1). A placement manager that implements the placement algorithm. To evaluate its scalability, we measured the time to place tenant requests in a datacenter with 100K machines. Over 100K representative requests, the median placement time is 4.13ms with a 99^{th} percentile of 2.72 seconds. Note that such placement only needs to be run when a tenant is admitted.

(2). For bandwidth allocation, we have implemented an extended version of RCP (RCP_w) that distributes network bandwidth in a weighted fashion, and is used for the hypervisor-to-hypervisor flows. This involves an end-host component and a switch component.

Ideally, the end-host component should run inside the hypervisor. For ease of prototyping, our implementation resides in user space. Application packets are intercepted and tunneled inside RCP_w flows with a custom header. We have a kernel driver that binds to the Ethernet interface and efficiently marshals packets between the NIC and the user space RCP_w stack. The switch is implemented on a server-grade PC and implements a store and forward architecture. It uses the same kernel driver to pass all incoming packets to a user space process.

In our implementation, switches allocate rate to flows once every round trip time. To keep switch overhead low, we use integer arithmetic for all rate calculations. Although each packet traverses the user-kernel space boundary, we can sustain four 1Gbps links at full duplex line rate. Futher, experiments in the next section show that we can achieve a link utilization of 96%. Overall, we find that our prototype imposes minimal overhead on the forwarding path.

4.4 Design discussion

Other placement goals. Today, placement of VMs in datacenters is subject to many constraints like their CPU and memory requirements, ensuring fault tolerance, energy efficiency and even reducing VM migrations [30]. Production placement managers like SCVMM [31] use heuristics to meet these constraints. Our flow network formulation maps tenant network requirements to constraints regarding VM placement which can be added to the set of input constraints used by existing placement managers. We defer an exploration of such extensions to future work. We do note that our constraints can be at odds with existing requirements. For example, while bandwidth guarantees entail placement locality, fault tolerance requires VMs be placed in different fault domains.

Hose-compliant allocation. Allocating bandwidth in proportion to the least paying of communication partners has implications for provisioning of cloud services. A service provider willing to pay for VMs with higher minimum bandwidth (and higher weight) will only improve the performance of its flows that are bottlenecked by the weight contributed by the service VMs. Performance for flows bottlenecked by client VMs will not improve. This is akin to network performance across the Internet today. When clients with poor last mile connectivity access a well-provisioned Internet service, their network performance is limited by their own capacity. Allocating bandwidth based on the sum of payments of communicating partners avoids this but at the expense of allowing network abuse.

5 Evaluation

We deployed our prototype implementation across a small testbed comprising twelve end-hosts arranged across four racks. Each rack has a top-of-rack (ToR) switch, and the ToR switches are connected through a root switch. All switches and end-hosts are Dell T3500 servers with a quad core Intel Xeon 2.27GHz processor, 4GB RAM and 1 Gbps interfaces, running Windows Server 2008 R2. Given our focus on network performance, the tenants are not actually allocated VMs but simply run as a user process. With 8 VM slots per host, the testbed has a total of 96 slots. We complement the testbed experiments with large-scale simulations. For this, we developed a simulator that models a multi-tenant datacenter with a three tier network topology. All simulation results here are based on a datacenter with 16K hosts and 4 VMs per host, resulting in 64K VMs. The network has an oversubscription of 1:10.

Overall, our evaluation covers three main aspects: (i) We combine testbed and simulation experiments to illustrate that Hadrian, by ensuring minimum VM bandwidth, benefits both tenants and providers, (ii) We use simulations to quantify the benefits of relaxing bandwidth guarantee semantics, and (iii) We use testbed experiments to show hose-compliance mitigates aggressive behavior in inter-tenant settings.

5.1 Cloud emulation experiments

We emulate the operation of a cloud datacenter on our testbed (and in the simulator) as follows. We generate

Placement → B/w Allocation	Greedy	Dependency -aware	**Hadrian's placement**
Per-flow	*Baseline*	*Baseline+*	–
Hose-compliant	–	–	**Hadrian**
Reservations	–	–	Oktopus [11]
Per-source	*Seawall* [12]	–	*Seawall*
PS-L	*FairCloud* [14]	–	*FairCloud*

Table 1: Solution space for cloud network sharing

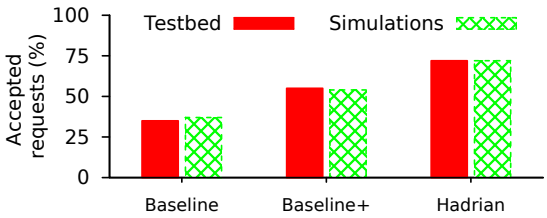


Figure 6: Accepted requests in testbed and simulator.

a synthetic workload with tenant requests arriving over time. A placement algorithm is used to allocate the requested VMs and if the request cannot be placed, it is rejected. The arrival of tenants is a Poisson process. By varying the rate at which tenants arrive, we control the *target VM occupancy* of the datacenter. This is the fraction of datacenter VMs that, on average, are expected to be occupied. As for bandwidth guarantees, tenants can choose from three classes for their minimum bandwidth– 50, 150 and 300 Mbps. By varying the fraction of tenant requests in each class, we control the *average minimum bandwidth* for tenants.

Tenants. We model two kinds of tenants– service tenants that have a wildcard (*) communication dependency and client tenants that depend on zero or more service tenants. Tenants request both VMs (V) and a minimum bandwidth (B^{min}). Each tenant runs a job involving network flows; some of these flows are intra-tenant while others are to VMs of service tenants. A job finishes when its flows finish. The fraction $F \in [0,1]$ of a tenant's flows that are inter-tenant allows us to determine the minimum bandwidth required by the tenant for inter-tenant communication. Overall, each tenant request is characterized by $<V, B^{min}, V*B^{min}*F$, *dependencies*$>$.

By abstracting away non-network resources, this simple workload model allows us to directly compare various network sharing approaches. While the workload is synthetic, we use our datacenter measurements to ensure it is representative of today's datacenters. For instance, the fraction of client tenants with dependencies (20%), the average number of dependencies (2), the fraction of inter-tenant flows (10-40%), and other workload parameters are as detailed in §2.1.

In the following sections, we compare Hadrian against alternate network sharing solutions. Since a complete network sharing framework ought to include both careful VM placement and bandwidth allocation, we consider various state of the art solutions for both–

VM Placement. We experiment with three placement approaches. (i) With *Greedy* placement, a tenant's VMs are greedily placed close to each other. (ii) With *Dependency-aware placement*, a tenant's VMs are placed close to each other and to VMs of existing tenants that the tenant has a dependency on. (iii) Hadrian's placement, described in §4.1, which is aware of tenant minimum bandwidths and their dependencies.

Bandwidth allocation. Apart from Per-flow, Per-source and PS-L sharing, we evaluate two other policies. (i) With hose-compliant allocation, bandwidth is allocated as described in §3.2. (ii) With "Reservations", VMs get to use their guaranteed bandwidth but no more. Hence, this allocation policy is not work conserving.

Table 1 summarizes the solution space for cloud network sharing. Note that by combining Hadrian's placement with Reservations, we can extend Oktopus [11] to inter-tenant settings. We begin by focussing on the approaches in the first two rows. The approach of placing tenant VMs greedily combined with the Per-flow sharing of the network reflects the operation of today's datacenters, and is thus used as a *Baseline* for comparison.

5.1.1 Testbed experiments

The experiment involves the arrival and execution of 100 tenant jobs on our testbed deployment. The average minimum bandwidth for tenants is 200 Mbps and requests arrive such that target VM occupancy is 75%. Note that operators like Amazon EC2 target an average occupancy of 70-80% [32]. Since our prototype uses weighted RCP, we emulate various bandwidth allocation policies by setting flow weights appropriately; for example, for Per-flow allocation, all flows have the same weight. Figure 6 shows that *Baseline* only accepts 35% of the requests, *Baseline+* accepts 55%, while Hadrian accepts 72%. This is despite the fact that Hadrian will reject a request if there is insufficient bandwidth while the other approaches will not. To show that Hadrian leads to comparable benefits at datacenter scale, we rely on large-scale simulations.

However, we first validate the accuracy of our simulator. To this end, we replayed the same set of jobs in the simulator. The figure also shows that the percentage of accepted requests in the testbed is similar to those accepted in the simulator; the difference ranges from 0-2%. Further, the completion time for 87% of the requests is the same across the testbed and simulator; at the 95^{th} percentile, requests are 17% faster in the simulator. This is because the simulator achieves perfect network sharing (e.g., no overheads). This gives us confidence in the fidelity of the simulation results below.

5.1.2 Large-scale simulations

We simulate a stream of 25K jobs on a datacenter with 16K servers. Figure 7(left) shows that, with Hadrian, the

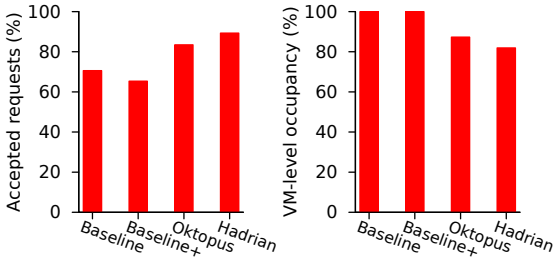


Figure 7: Provider can accept more requests with Hadrian. (average B^{min} = 200 Mbps)

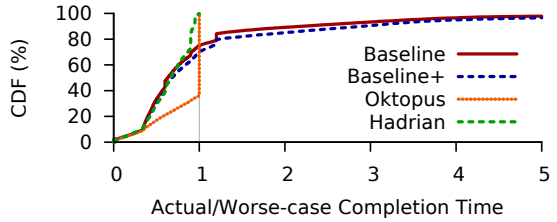


Figure 8: With *Baseline* (and *Baseline+*), many tenants receive poor network performance and finish past the worst-case completion time estimate.

provider is able to accept 20% more requests than both *Baseline* and *Baseline+*. We also simulate the use of hard reservations for allocating bandwidth (Oktopus), and find that Hadrian can still accept 6% more requests. Further, figure 7(right) shows the average VM occupancy during the experiment. With Hadrian, the average VM utilization is 87% as compared to 99.9% with *Baseline* and 90% with Oktopus. This is because jobs finish earlier with Hadrian. Thus, Hadrian allows the provider to accommodate more requests while reducing the VM-level utilization which, in turn, allows more future requests to be accepted.

To understand this result, we examine the performance of individual requests. Since tenants are associated with minimum bandwidths, each tenant can estimate the worst-case completion time for its flows and hence, its job. Figure 8 shows the CDF for the ratio of a job's actual completion time to the worst-case estimate. With Hadrian, all requests finish before the worst-case estimate. With Oktopus, tenants get their requested bandwidth but no more, so most requests finish at the worst-case estimate.[3] As a contrast, with *Baseline*, many tenants get very poor network performance. The completion time for 15% tenants is 1.25x the worst-case estimate and for 5% tenants, it is 3.4x the worst-case. These outliers occupy VMs longer, thus driving up utilization but reducing actual throughput. By ensuring minimum bandwidth for VMs and thus avoiding such outliers, Hadrian allows the provider to accept more requests.

[3]Requests that finish earlier with Oktopus have all their flows between co-located VMs in the same physical machine, hence achieving bandwidth greater than their reservation, so the job can finish early.

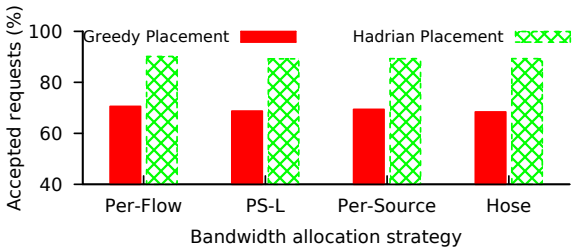


Figure 9: With non-aggressive tenants, Hadrian's placement provides most of the gains.

Beyond this, we also experimented with other values for the simulation parameters– the average minimum bandwidth, the target occupancy, the network oversubscription and the percentage of inter-tenant traffic. The results are along expected lines so we omit them for brevity but they are available in [27]. For example, Hadrian's gains increase as inter-tenant traffic increases and network oversubscription increases. Further, Hadrian can offer benefits even when there is no oversubscription. While it accepts the same number of requests as *Baseline*, 22% of requests are outliers with *Baseline*.

Cost analysis. Today's cloud providers charge tenants a fixed amount per hour for each VM; for instance, Amazon EC2 charges $0.08/hr for small VMs. Hence, the improved tenant performance with Hadrian has implications for their cost too. For the experiment above, the average tenant would pay 34% less with Hadrian than *Baseline*. This is because there are no outliers receiving very poor network performance. From the provider's perspective though, there are two competing factors. Hadrian allows them to accommodate more tenants but tenants finish faster and hence, pay less. We find the provider's revenue with Hadrian is 82% of that with *Baseline*. This reduction in revenue can be overcome by new pricing models that account for the added value Hadrian offers. Since Hadrian provides tenants with VMs that have minimum bandwidth, the provider can increase the price of VMs. We repeat the cost analysis to determine how much tenants would have to pay so that the provider remains revenue neutral and find that the average tenant would still pay 19% less. Overall, the results above show that Hadrian allows the provider to offer network guarantees while reducing the average tenant cost.

Importance of relaxed guarantee semantics. In the experiments above, we find that with simple hose guarantees where all tenants are assumed to speak to each other, the provider is only able to accept 1% of the requests! However, when the provider is aware of tenant dependencies, it can accept 85% of the requests. This is because bandwidth constraints need to enforced on far fewer links. Hierarchical guarantees allow the provider to accept a further 5% requests. These results highlight the importance of relaxed guarantee semantics.

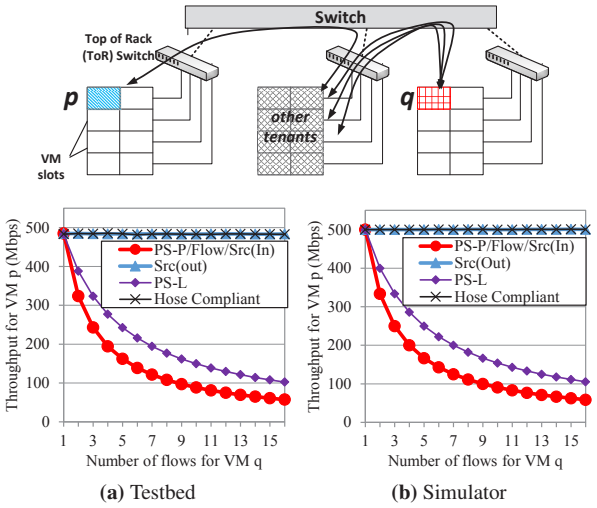


Figure 10: By sending and receiving more flows, q can degrade p's network performance.

Importance of placement. We now also consider the Per-source and PS-L allocation, and compare their performance, when coupled with Greedy and Hadrian's placement. Figure 9 shows that Hadrian's placement provides gains irrespective of how bandwidth is being allocated. Here, the allocation policy does not have much impact because *all tenants have the same traffic pattern*. Hence, under scenarios with well-behaved tenants, VM placement dictates the datacenter throughput. However, as we show in the next section, when some tenants are aggressive, the allocation policy does matter.

5.1.3 Benefits of hose-compliant allocation

To illustrate the benefits of hose-compliance relative to alternate bandwidth allocation policies, we focus on a simple scenario that captures aggressive tenant behavior. The experiment involves two tenants, each with one VM (p and q). As shown in Figure 10 (top), VM p has one flow while VM q has a lot of flows to VMs of other tenants. All flows are bottlenecked at the same link. VM q could be acting maliciously and initiating multiple flows to intentionally degrade p's network performance. Alternatively, it could just be running a popular service that sends or receives a lot of traffic which, in turn, can hurt p's performance. All VMs, including destination VMs, have a minimum bandwidth of 300 Mbps.

Figures 10a and 10b show the average bandwidth for VM p on the testbed and in the simulator respectively. With hose-compliant allocation, VM p's bandwidth remains the same throughout. As a contrast, with other policies, p's bandwidth degrades as q has more flows. This is because, with these approaches, there is no limit on the fraction of a link's bandwidth a VM can grab. This allows tenants to abuse the network at the expense of others. By bounding tenant impact, hose-compliant allocation addresses this. Note that with Per-source allocation, p's bandwidth does not degrade if both VMs are sending traffic (labelled as "out") but it does if they are receiving traffic (labelled as "in"). Instead, hose-compliant allocation is not impacted by the traffic direction.

The figures also show that the testbed results closely match simulation results for all policies. With hose-compliance, VM p's bandwidth on the testbed is 480 Mbps as compared to the ideal 500 Mbps. This shows our weighted RCP implementation is performant and can achieve a link utilization of 96%.

6 Related work

Many recent efforts tackle the cloud network sharing problem. They propose different sharing policies, including reservations [9,11], time-varying reservations [33], minimum bandwidth reservations [10,14,34], per-source fairness [12] and per-tenant fairness [13]. Mogul et al. [35] present a useful survey of these proposals. However, as detailed in this paper, none of these proposals explicitly target inter-tenant communication which poses its own challenges.

To achieve desirable network sharing, we have borrowed and extended ideas from many past proposals. The hose model [20] has been used to capture both reservations [11] and minimum bandwidth guarantees [14,21]. We extend it by adding communication dependencies and hierarchy. The use of hierarchy means that Hadrian offers aggregate, "per-tenant" minimum guarantees for inter-tenant traffic. This is inspired by Oktopus [11] and NetShare [13] that offer per-tenant reservations and weights respectively. CloudPolice [3] argues for network access control in inter-tenant settings. However, many cloud services today are open to tenants. Hence, network access control needs to be coupled with a robust network sharing mechanism like Hadrian.

7 Concluding remarks

Inter-tenant communication plays a critical role in today's datacenters. In this paper, we show this necessitates a rethink of how the cloud network is shared. To ensure provider flexibility, we modify the kind of bandwidth guarantees offered to tenants. To ensure robust yet proportional network sharing, we argue for coupling the maximum bandwidth allocated to tenants to their payment. Tying these ideas together, we propose Hadrian, a network sharing framework that uses hose-compliant allocation and bandwidth-aware VM placement to achieve desirable network sharing properties for both intra- and inter-tenant communication. Our evaluation shows that Hadrian's mechanisms are practical. Further, apart from improving the performance of both tenants and providers, it ensures robust network sharing.

References

[1] "Amazon AWS Products," http://aws.amazon.com/products/.

[2] "Amazon AWS Marketplace," http://aws.amazon.com/marketplace/.

[3] L. Popa, M. Yu, S. Ko, S. Ratnasamy, and I. Stoica, "CloudPolice: Taking access control out of the network," in *Proc. of ACM HotNets*, 2010.

[4] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: comparing public cloud providers," in *Proc. of ACM IMC*, 2010.

[5] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," in *Proc. of VLDB*, 2010.

[6] "Measuring EC2 performance," http://tech.mangot.com/roller/dave/entry/ec2_variability_the_numbers_revealed.

[7] A. Iosup, N. Yigitbasi, and D. Epema, "On the Performance Variability of Cloud Services," Delft University, Tech. Rep. PDS-2010-002, 2010.

[8] E. Walker, "Benchmarking Amazon EC2 for high performance scientific computing," *Usenix Login*, vol. 33, October 2008.

[9] 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 *Proc. of ACM CoNEXT*, 2010.

[10] P. Soares, J. Santos, N. Tolia, and D. Guedes, "Gatekeeper: Distributed Rate Control for Virtualized Datacenters," HP Labs, Tech. Rep. HP-2010-151, 2010.

[11] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards Predictable Datacenter Networks," in *Proc. of ACM SIGCOMM*, 2011.

[12] A. Shieh, S. Kandula, A. Greenberg, and C. Kim, "Sharing the Datacenter Network," in *Proc. of Usenix NSDI*, 2011.

[13] T. Lam, S. Radhakrishnan, A. Vahdat, and G. Varghese, "NetShare: Virtualizing Data Center Networks across Services," University of California, San Diego, Tech. Rep. CS2010-0957, May 2010.

[14] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "FairCloud: Sharing the Network In Cloud Computing," in *Proc of ACM SIGCOMM*, 2012.

[15] "Windows Azure Pricing," http://www.windowsazure.com/en-us/pricing/details.

[16] "Amazon EC2 Pricing," http://aws.amazon.com/ec2/pricing/.

[17] "Amazon Case Studies," http://aws.amazon.com/solutions/case-studies/.

[18] D. Ghoshal, R. S. Canon, and L. Ramakrishnan, "I/O performance of virtualized cloud environments," in *Proc. of DataCloud-SC*, 2011.

[19] Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez, and M. Humphrey, "Early observations on the performance of Windows Azure," in *Proc. of HPDC*, 2010.

[20] 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 *Proc. of ACM SIGCOMM*, 1999.

[21] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "The Price Is Right: Towards Location-independent Costs in Datacenters," in *Proc. of ACM HotNets*, 2011.

[22] H. Ballani, Y. Chawathe, S. Ratnasamy, T. Roscoe, and S. Shenker, "Off by Default!" in *Proc. of ACM HotNets*, 2005.

[23] S. Lee, R. Panigrahy, V. Prabhakaran, V. Ramasubramanian, K. Talwar, L. Uyeda, and U. Wieder, "Validating Heuristics for Virtual Machines Consolidation," MSR, Tech. Rep. MSR-TR-2011-9, 2011.

[24] 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 *Proc. of ACM SIGCOMM*, 2009.

[25] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. of ACM SIGCOMM*, 2008.

[26] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks," in *Proc. of Usenix NSDI*, 2010.

[27] H. Ballani, D. Gunawardena, and T. Karagiannis, "Network Sharing in Multi-tenant Datacenters," MSR, Tech. Rep. MSR-TR-2012-39, 2012.

[28] N. Dukkipati, "Rate Control Protocol (RCP): Congestion control to make flows complete quickly," Ph.D. dissertation, Stanford University, 2007.

[29] D. Katabi, M. Handley, and C. Rohrs, "Congestion Control for High Bandwidth-Delay Product Networks," in *Proc. of ACM SIGCOMM*, Aug. 2002.

[30] A. Rai, R. Bhagwan, and S. Guha, "Generalized Resource Allocation for the Cloud," in *Proc of ACM SOCC*, 2012.

[31] "Microsoft System Center Virtual Machine Manager," http://www.microsoft.com/en-us/server-cloud/system-center/default.aspx.

[32] "Amazon's EC2 Generating 220M," http://cloudscaling.com/blog/cloud-computing/amazons-ec2-generating-220m-annually.

[33] D. Xie, N. Ding, Y. C. Hu, and R. Kompella, "The Only Constant is Change: Incorporating Time-Varying Network Reservations in Data Centers," in *Proc of ACM SIGCOMM*, 2012.

[34] V. Jeyakumar, M. Alizadeh, D. Mazires, B. Prabhakar, and C. Kim, "EyeQ: Practical Network Performance Isolation for the Multi-tenant Cloud," in *Proc. of Usenix HotCloud*, 2012.

[35] J. C. Mogul and L. Popa, "What we talk about when we talk about cloud network performance," *ACM CCR*, Oct. 2012.

Effective Straggler Mitigation: Attack of the Clones

Ganesh Ananthanarayanan[1], Ali Ghodsi[1,2], Scott Shenker[1], Ion Stoica[1]

[1] *University of California, Berkeley* [2] *KTH/Sweden*

{ganesha,alig,shenker,istoica}@cs.berkeley.edu

Abstract

Small jobs, that are typically run for interactive data analyses in datacenters, continue to be plagued by disproportionately long-running tasks called *stragglers*. In the production clusters at Facebook and Microsoft Bing, even after applying state-of-the-art straggler mitigation techniques, these latency sensitive jobs have stragglers that are on average 8 times slower than the median task in that job. Such stragglers increase the average job duration by 47%. This is because current mitigation techniques all involve an element of waiting and speculation. We instead propose full *cloning* of small jobs, avoiding waiting and speculation altogether. Cloning of small jobs only marginally increases utilization because workloads show that while the majority of jobs are small, they only consume a small fraction of the resources. The main challenge of cloning is, however, that extra clones can cause contention for intermediate data. We use a technique, *delay assignment*, which efficiently avoids such contention. Evaluation of our system, Dolly, using production workloads shows that the small jobs speedup by 34% to 46% *after* state-of-the-art mitigation techniques have been applied, using just 5% extra resources for cloning.

1 Introduction

Cloud computing has achieved widespread adoption due to its ability to automatically parallelize a *job* into multiple short *tasks*, and transparently deal with the challenge of executing these tasks in a distributed setting. One such fundamental challenge is *straggling tasks*, which is faced by all cloud frameworks, such as MapReduce [1], Dryad [2], and Spark [3]. Stragglers are tasks that run much slower than other tasks, and since a job finishes only when its last task finishes, stragglers delay job completion. Stragglers especially affect *small jobs*, *i.e.,* jobs that consist of a few tasks. Such jobs typically get to run all their tasks at once. Therefore, even if a single task is slow, *i.e.,* straggle, the whole job is significantly delayed.

Small jobs are pervasive. Conversations with datacenter operators reveal that these small jobs are typically used when performing interactive and exploratory analyses. Achieving low latencies for such jobs is critical to enable data analysts to efficiently explore the search space. To obtain low latencies, analysts already restrict their queries to small but carefully chosen datasets, which results in jobs consisting of only a few short tasks. The trend of such exploratory analytics is evident in traces we have analyzed from the Hadoop production cluster at Facebook, and the Dryad cluster at Microsoft Bing. Over 80% of the Hadoop jobs and over 60% of the Dryad jobs are small with fewer than ten tasks[1]. Achieving low latencies for these small interactive jobs is of prime concern to datacenter operators.

The problem of stragglers has received considerable attention already, with a slew of *straggler mitigation* techniques [1, 4, 5] being developed. These techniques can be broadly divided into two classes: *black-listing* and *speculative execution*. However, our traces show that *even after* applying state-of-the-art blacklisting and speculative execution techniques, the small jobs have stragglers that, on average, run eight times slower than that job's median task, slowing them by 47% on average. Thus, stragglers remain a problem for small jobs. We next explain the limitations of these two approaches.

Blacklisting identifies machines in bad health (*e.g.,* due to faulty disks) and avoids scheduling tasks on them. The Facebook and Bing clusters, in fact, blacklist roughly 10% of their machines. However, stragglers occur on the non-blacklisted machines, often due to intrinsically complex reasons like IO contentions, interference by periodic maintenance operations and background services, and hardware behaviors [6].

For this reason, speculative execution [1, 4, 5, 7] was explored to deal with stragglers. Speculative execution *waits* to observe the progress of the tasks of a job and launches duplicates of those tasks that are slower. However, speculative execution techniques have a fundamental limitation when dealing with small jobs. Any meaningful comparison requires waiting to collect statistically significant samples of task performance. Such waiting limits their agility when dealing with stragglers in small jobs as they often start all their tasks simultaneously. The problem is exacerbated when some tasks start straggling when they are well into their execution. Spawning a speculative copy at that point might be too late to help.

In this paper, we propose a different approach. Instead of waiting and trying to predict stragglers, we take speculative execution to its extreme and propose launching multiple *clones* of every task of a job and only use the result of the clone that finishes first. This technique is both general and robust as it eschews waiting, speculating, and finding complex correlations. Such *proactive*

[1] The length of a task is mostly invariant across small and large jobs.

cloning will significantly improve the agility of straggler mitigation when dealing with small interactive jobs.

Cloning comes with two main challenges. The first challenge is that extra clones might use a prohibitive amount of extra resources. However, our analysis of production traces shows a strong *heavy-tail distribution* of job sizes: the smallest 90% of jobs consume as less as 6% of the resources. The interactive jobs whose latency we seek to improve all fall in this category of small jobs. We can, hence, improve them by using few extra resources.

The second challenge is the potential *contention* that extra clones create on intermediate data, possibly hurting job performance. Efficient cloning requires that we clone each task and use the output from the clone of the task that finishes first. This, however, can cause contention for the intermediate data passed between tasks of the different phases (*e.g.,* map, reduce, join) of the job; frameworks often compose jobs as a graph of *phases* where tasks of downstream phases (*e.g.,* reduce) read the output of tasks of upstream phases (*e.g.,* map). If all downstream clones read from the upstream clone that finishes first, they contend for the IO bandwidth. An alternate that avoids this contention is making each downstream clone read exclusively from only a single upstream clone. But this staggers the start times of the downstream clones.

Our solution to the contention problem, *delay assignment*, is a hybrid solution that aims to get the best of both the above pure approaches. It is based on the intuition that most clones, except few stragglers, finish nearly simultaneously. Using a cost-benefit analysis that captures this small variation among the clones, it checks to see if clones can obtain exclusive copies before assigning downstream clones to the available copies of upstream outputs. The cost-benefit analysis is generic to account for different communication patterns between the phases, including all-to-all (MapReduce), many-to-one (Dryad), and one-to-one (Dryad and Spark).

We have built Dolly, a system that performs cloning to mitigate the effect of stragglers while operating within a resource budget. Evaluation on a 150 node cluster using production workloads from Facebook and Bing shows that Dolly improves the average completion time of the small jobs by 34% to 46%, respectively, with LATE [5] and Mantri [4] as baselines. These improvements come with a resource budget of merely 5% due to the aforementioned heavy-tail distribution of job-sizes. By picking the fastest clone of every task, Dolly effectively reduces the slowest task from running $8\times$ slower on average to $1.06\times$, thus, effectively eliminating all stragglers.

2 The Case for Cloning

In this section we quantify: (i) magnitude of stragglers and the potential in eliminating them, and (ii) power law distribution of job sizes that facilitate aggressive cloning.

	Facebook	Microsoft Bing
Dates	Oct 2010	May-Dec* 2009
Framework	Hadoop	Dryad
File System	HDFS [9]	Cosmos
Script	Hive [10]	Scope [11]
Jobs	375K	200K
Cluster Size	3,500	Thousands
Straggler–mitigation	LATE [5]	Mantri [4]

* *One week in each month*

Table 1: **Details of Facebook and Bing traces.**

Production Traces: Our analysis is based on traces from Facebook's production Hadoop [8] cluster and Microsoft Bing's production Dryad [2] cluster. These are large clusters with thousands of machines running jobs whose performance and output have significant impact on productivity and revenue. Therefore, each of the machines in these clusters is well-provisioned with tens of cores and sufficient (tens of GBs) memory. The traces capture the characteristics of over half a million jobs running across many months. Table 1 lists the relevant details of the traces. The Facebook cluster employs the LATE straggler mitigation strategy [5], while the Bing cluster uses the Mantri straggler mitigation strategy [4].

2.1 Stragglers in Jobs

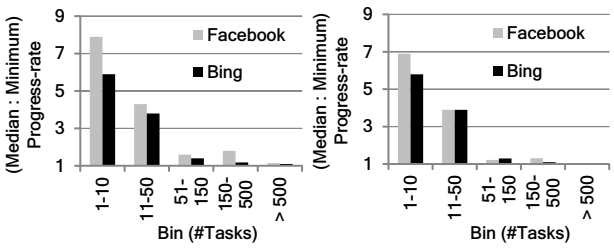
We first quantify the magnitude and impact of stragglers, and then show that simple blacklisting of machines in the cluster is insufficient to mitigate them.

2.1.1 Magnitude of Stragglers and their Impact

A job consists of a graph of *phases* (*e.g.,* map, reduce, and join), with each phase executing the same type of tasks in parallel. We identify stragglers by comparing the *progress rates* of tasks within a phase. The progress rate of a task is defined as the size of its input data divided by its duration. In absence of stragglers, progress rates of tasks of a phase are expected to be similar as they perform similar IO and compute operations. We use the progress rate instead of the task's duration to remain agnostic to skews in work assignment among tasks [4]. Techniques have been developed to deal with the problem of data skews among tasks [12, 13, 14] and our approach is complementary to those techniques.

Within each phase, we measure the *slowdown ratio*, *i.e.,* the ratio of the progress rate of the median task to the slowest task. The negative impact of stragglers increases as the slowdown ratio increases. We measure the slowdown ratio after applying the LATE and Mantri mitigations; a what-if simulation is used for the mitigation strategy that the original trace did not originally deploy.

Figure 1a plots the slowdown ratio by binning jobs according to their number of tasks, with LATE in effect.



(a) LATE (b) Mantri

Figure 1: **Slowdown ratio *after* applying LATE and Mantri. Small jobs see a higher prevalence of stragglers.**

p_b	Blacklisted Machines (%)		Job Improvement (%)	
	5 min	1 hour	5 min	1 hour
0.3	4%	6%	7.1%	8.4%
0.5	1.6%	2.8%	4.4%	5.2%
0.7	0.8%	1.2%	2.3%	2.8%

Table 2: **Blacklisting by predicting straggler probability. We show the fraction of machines that got blacklisted and the improvements in completion times by avoiding them.**

Phases in jobs with fewer than ten tasks, have a median value of this ratio between 6 and 8, *i.e.,* the slowest task is up to $8\times$ slower than the median task in the job. Also, small jobs are hit harder by stragglers.[2] This is similar even if Mantri [4] was deployed. Figure 1b shows that the slowest task is still $7\times$ slower than the median task, with Mantri. However, both LATE and Mantri effectively mitigate stragglers in large jobs.

Speculation techniques are not as effective in mitigating stragglers in small jobs as they are with large jobs because they rely on comparing different tasks of a job to identify stragglers. Comparisons are effective with more samples of task performance. This makes them challenging to do with small jobs because not only do these jobs have fewer tasks but also start all of them simultaneously.

Impact of Stragglers: We measure the potential in speeding up jobs in the trace using the following crude analysis: replace the progress rate of every task of a phase that is slower than the median task with the median task's rate. If this were to happen, the average completion time of jobs improves by 47% and 29% in the Facebook and Bing traces, respectively; small jobs (those with ≤ 10 tasks) improve by 49% and 38%.

2.1.2 Blacklisting is Insufficient

An intuitive solution for mitigating stragglers is to *blacklist* machines that are likely to cause them and avoid scheduling tasks on them. For this analysis, we classify a task as a straggler if its progress rate is less than half of the median progress rate among tasks in its phase. In our trace, stragglers are not restricted to a small set of machines but are rather spread out uniformly through the cluster. This is not surprising because both the clusters already blacklist machines with faulty disks and other hardware troubles using periodic diagnostics.

We enhance this blacklisting by monitoring machines at finer time intervals and employing temporal prediction techniques to warn about straggler occurrences. We use an EWMA to predict stragglers—the probability of a machine causing a straggler in a time window is equally dependent on its straggler probability in the previous window and its long-term average. Machines with a predicted straggler probability greater than a threshold (p_b) are blacklisted for that time window but considered again for scheduling in the next time window.

We try time windows of 5 minutes and 1 hour. Table 2 lists the fraction of machines that get blacklisted and the resulting improvement in job completion times by eliminating stragglers on them, in the Facebook trace. The best case eliminates only 12% of the stragglers and improves the average completion time by only 8.4% (in the Bing trace, 11% of stragglers are eliminated leading to an improvement of 6.2%). This is in harsh contrast with potential improvements of 29% to 47% if all stragglers were eliminated, as shown in §2.1.1.

The above results do not prove that effective blacklisting is impossible, but shows that none of the blacklisting techniques that we and, to our best knowledge, others [6] have tried effectively prevent stragglers, suggesting that such correlations either do not exist or are hard to find.

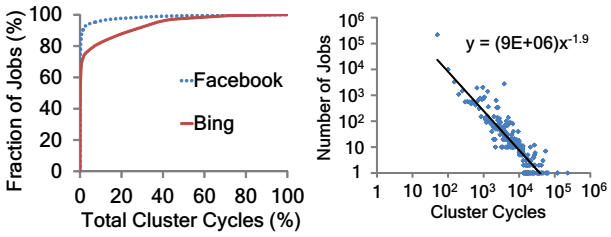
2.2 Heavy Tail in Job Sizes

We observed that smaller jobs are most affected by stragglers. These jobs were submitted by users for iterative experimental purposes. For example, researchers tune the parameters of new mining algorithms by evaluating it on a small sample of the dataset. For this reason, these jobs consist of just a few tasks. In fact, in both our traces, we have noted a correlation between a job's duration and the number of tasks it has, *i.e.,* jobs with shorter durations tend to have fewer tasks. Short and predictable response times for these jobs is of prime concern to datacenter operators as they significantly impact productivity.

On the one hand, small interactive jobs absolutely dominate the cluster and have stringent latency demands. In the Facebook and Bing traces, jobs with ≤ 10 tasks account for 82% and 61% of all the jobs, respectively. On the other hand, they are the most affected by stragglers.

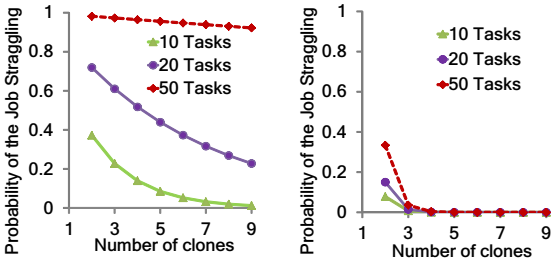
Despite this, we can clone all the small jobs using few extra resources. This is because job sizes have a *heavy-tail* distribution. Just a few large jobs consume most of

[2] Implicit in our explanation is that small interactive jobs consist of just a few tasks. While we considered alternate definitions based on input size and durations, in both our traces, we see a high correlation between jobs running for short durations and the number of tasks they contain along with the size of their input.



(a) Heavy-tail (b) Power Law

Figure 2: **Heavy tail. Figure (a) shows the heavy tail in the fraction of total resources used. Figure (b) shows that the distribution of cluster resources consumed by jobs, in the Facebook trace, follows a power law. Power-law exponents are** 1.9 **and** 1.8 **when fitted with least squares regression in the Facebook and Bing traces.**



(a) Job-level Cloning (b) Task-level Cloning

Figure 3: **Probability of a job straggling for varying number of clones, and sample jobs of** 10, 20 **and** 50 **tasks. Task-level cloning requires fewer clones than job-level cloning to achieve the same probability of the job straggling.**

the resources in the cluster, while the cluster is dominated by small interactive jobs. As Figure 2a shows, 90% of the smallest jobs consume only 6% and 11% of the total cluster resources in the Facebook and Bing clusters, respectively. Indeed, the distribution of resources consumed by jobs follows a power law (see Figure 2b). In fact, at any point in time, the small jobs do not use more than 2% of the overall cluster resources.

The heavy-tail distribution offers potential to speed up these jobs by using few extra resources. For instance, cloning each of the smallest 90% of the jobs three times increases overall utilization by merely 3%. This is well within reach of today's underutilized clusters which are heavily over-provisioned to satisfy their peak demand of over 99%, that leaves them idle at other times [15, 16].

Google recently released traces from their cluster job scheduler that schedules a mixed workload of MapReduce batch jobs, interactive queries and long-running services [17]. Analysis of these traces again reveal a heavy-tail distribution of job sizes, with 92% of the jobs accounting for only 2% of the overall resources [18].

3 Cloning of Parallel Jobs

We start this section by describing the high-level idea of cloning. After that (§3.1) we determine the granularity of cloning, and settle for cloning at the granularity of tasks, rather than entire jobs, as the former requires fewer clones. Thereafter (§3.2), we investigate the number of clones needed if we desire the probability of a job straggling to be at most ϵ, while staying within a cloning budget. Finally (§3.3), as we are unlikely to have room to clone every job in the cluster, we show a very simple admission control mechanism that decides when to clone jobs. An important challenge of cloning—handling data contention between clones—is dealt with in §4.

In contrast to *reactive* speculation solutions [1, 4, 5], Dolly advocates a *proactive* approach—straightaway launch multiple clones of a job and use the result of the first clone that finishes. Cloning makes straggler mitigation agile as it does not have to wait and observe a task before acting, and also removes the risk inherent in speculation—speculating the wrong tasks or missing the stragglers. Similar to speculation, we assume that picking the earliest clone does not bias the results, a property that generally holds for data-intensive computations.

3.1 Granularity of Cloning

We start with a job consisting of a single phase. A crucial decision affecting efficiency is the granularity of cloning. A simple option is to clone at the granularity of jobs. For every job submitted to the cluster, multiple clones of the entire job are launched. Results are taken from the earliest job that finishes. Such job-level cloning is appealing due to its simplicity and ease of implementation.

A fine-grained alternative is to clone at the granularity of individual tasks. Thus, multiple clones of each task are launched. We refer to the different clones of the same task as a *clone group*. In every clone group, we then use the result of the clone that finishes first. Therefore, unlike job-level cloning, task-level cloning requires internal changes to the execution engine of the framework.

As a result of the finer granularity, for the same number of clones, task-level cloning provides better probabilistic guarantees for eliminating stragglers compared to job-level cloning. Let p be the probability of a task straggling. For a single-phased job with n parallel tasks and c clones, the probability that it straggles is $(1-(1-p)^n)^c$ with job-level cloning, and $1-(1-p^c)^n$ with task-level cloning. Figure 3 compares these probabilities. Task-level cloning gains more per clone and the probability of the job straggling drops off faster.

Task-level cloning's resource efficiency is desirable because it reduces contention on the input data which is read from file systems like HDFS [9]. If replication of input data does not match the number of clones, the clones contend for IO bandwidth in reading the data. Increas-

ing replication, however, is difficult as clusters already face a dearth of storage space [19, 20]. Hence, due to its efficiency, we opt for task-level cloning in Dolly.

3.2 Budgeted Cloning Algorithm

Pseudocode 1 describes the cloning algorithm that is executed at the scheduler per job. The algorithm takes as input the cluster-wide probability of a straggler (p) and the acceptable risk of a job straggling (ϵ). We aim for an ϵ of 5% in our experiments. The probability of a straggler, p, is calculated every hour, where the straggler progresses at less than half the median task in the job. This coarse approach suffices for our purpose.

Dolly operates within an allotted resource budget. This budget is a configurable fraction (β) of the total capacity of the cluster (C). At no point does Dolly use more than this cloning budget. Setting a hard limit eases deployment concerns because operators are typically nervous about increasing the average utilization by more than a few percent. Utilization and capacity are measured in number of slots (computation units allotted to tasks).

The pseudocode first calculates the desired number of clones per task (step 2). For a job with n tasks, the number of clones desired by task-level cloning, c, can be derived to be at least $\log\left(1-(1-\epsilon)^{(1/n)}\right)/\log p$. [3] The number of clones that are eventually spawned is limited by the resource budget ($C \cdot \beta$) and a utilization threshold (τ), as in step 3. The job is cloned only if there is room to clone all its tasks, a policy we explain shortly in §3.3. Further, cloning is avoided if the cluster utilization after spawning clones is expected to exceed a ceiling τ. This ceiling avoids cloning during heavily-loaded periods.

Note that Pseudocode 1 spawns the same number of clones to all the tasks of a job. Otherwise, tasks with fewer clones are more likely to lag behind. Also, there are no conflicts between jobs in updating the shared variables B_U and U because the centralized scheduler handles cloning decisions one job at a time.

Multi-phased Jobs: For multi-phased jobs, Dolly uses Pseudocode 1 to decide the number of clones for tasks of *every* phase. However, the number of clones for tasks of a downstream phase (*e.g.,* reduce) never exceeds the number of clones launched its upstream phase (*e.g.,* map). This avoids contention for intermediate data (we revisit this in §4). In practice, this limit never applies because small jobs have equal number of tasks across their phases. In both our traces, over 91% of the jobs with ≤ 10 tasks have equal number of tasks in their phases.

3.3 Admission Control

The limited cloning budget, β, should preferably be utilized to clone the small interactive jobs. Dolly achieves this using a simple policy of *admission control*.

```
1: procedure CLONE(n tasks, p, ε)
       C: Cluster Capacity, U: Cluster Utilization
       β: Budget in fraction, B_U: Utilized budget in #slots
2:     c = ⌈log(1 − (1 − ε)^(1/n)) / log p⌉
3:     if (B_U + c · n) ≤ (C · β) and (U + c · n) ≤ τ then
                ▷ Admission Control: Sufficient capacity to
                  create c clones for each task
4:         for each task t do
               Create c clones for t
           B_U ← B_U + c · n
```

Pseudocode 1: **Task-level cloning for a single-phased job with n parallel tasks, on a cluster with probability of straggler as p, and the acceptable risk of straggler as ϵ.**

Whenever the first task of a job is to be executed, the admission control mechanism computes, as previously explained, the number of clones c that would be required to reach the target probability ϵ of that job straggling. If, at that moment, there is room in the cloning budget for creating c copies of all the tasks, it admits cloning the job. If there is not enough budget for c clones of all the tasks, the job is simply denied cloning and is executed without Dolly's straggler mitigation. The policy of admission control implicitly biases towards cloning small jobs—the budget will typically be insufficient for creating the required number of clones for the larger jobs. Step 3 in Pseudocode 1 implements this policy.

Many other competing policies are possible. For instance, a job could be partially cloned if there is not enough room for c clones. Furthermore, preemption could be used to cancel the clones of an existing job to make way for cloning another job. It turns out that these competing policies buy little performance compared to our simple policy. We compare these policies in §5.5.

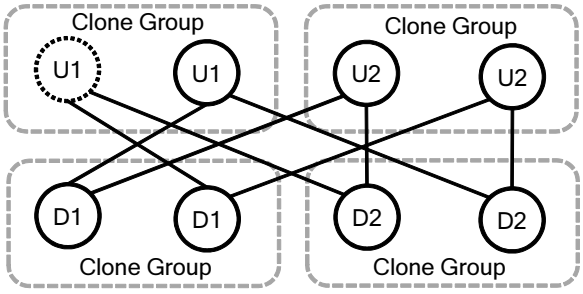
4 Intermediate Data Access with Dolly

A fundamental challenge of cloning is the potential contention it creates in reading data. Downstream tasks in a job read intermediate data from upstream tasks according to the communication pattern of that phase (all-to-all, many-to-one, one-to-one). The clones in a downstream clone group would ideally read their intermediate data from the upstream clone that finishes first as this helps them all start together.[4] This, however, can create contention at the upstream clone that finishes first. Dealing with such contentions is the focus of this section.

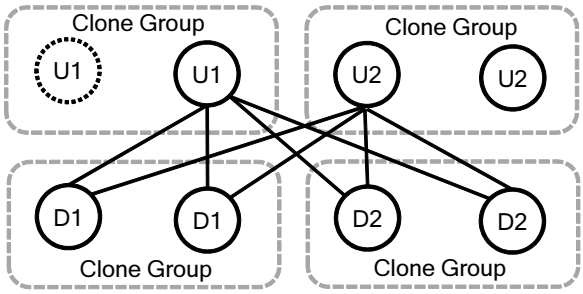
We first (§4.1) explore two pure strategies at opposite ends of the spectrum for dealing with intermediate data contention. At one extreme, we completely avoid con-

[3] The probability of a job straggling can be at most ϵ, *i.e.,* $1-(1-p^c)^n \leq \epsilon$. The equation is derived by solving for c.

[4] Intermediate data typically only exists on a single machine, as it is not replicated to avoid time and resource overheads. Some systems do replicate intermediate data [4, 21] for fault-tolerance but limit this to replicating only a small fraction of the data.



(a) Contention-Avoidance Cloning (CAC)



(b) Contention Cloning (CC)

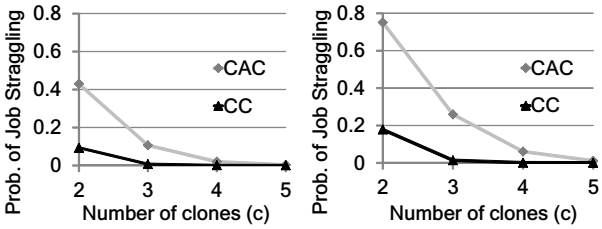
Figure 4: **Intermediate data contention. The example job contains two upstream tasks (U1 and U2) and two downstream tasks (D1 and D2), each cloned twice. The clone of U1 is a straggler (marked with a dotted circle). CAC waits for the straggling clone while CC picks the earliest clone.**

tention by assigning each upstream clone, as it finishes, to a new downstream task clone. This avoids contention because it guarantees that every upstream task clone only transfers data to a single clone per downstream clone group. At another extreme, the system ignores the extra contention caused and assumes that the first finished upstream clone in every clone group can sustain transferring its intermediate output to all downstream task clones. As we show (§4.2), the latter better mitigates stragglers compared to the former strategy. However, we show (§4.3) that the latter may lead to congestion whereas the former completely avoids it. Finally (§4.4), we settle on a hybrid between the two (§4.4), *delay assignment* that far outperforms these two pure strategies.

4.1 Two Opposite Strategies

We illustrate two approaches at the opposite ends of the spectrum through a simple example. Consider a job with two phases (see Figure 4) and an all-to-all (*e.g.,* shuffle) communication pattern between them (§4.4 shows how this can be generalized to other patterns). Each of the phases consist of two tasks, and each task has two clones.

The first option (Figure 4a), which we call Contention-Avoidance Cloning (CAC) eschews contention altogether. As soon as an upstream task clone finishes, its output is sent to exactly one downstream task clone per clone group. Thus, the other downstream task clones have to wait for another upstream task clone to finish before they can start their computation. We call this Contention-Avoidance Cloning (CAC). Note that in CAC an upstream clone will send its intermediate data to the exact same number of other tasks as if no cloning was done, avoiding contention due to cloning. The disadvantage with CAC is that when some upstream clones straggle, the corresponding downstream clones that read data from them automatically lag behind.



(a) $n = 10$ tasks (b) $n = 20$ tasks

Figure 5: **CAC vs. CC: Probability of a job straggling.**

The alternate option (Figure 4b), Contention Cloning (CC), alleviates this problem by making all the tasks in a downstream clone group read the output of the upstream clone that finishes first. This ensures that no downstream clone is disadvantaged, however, all of them may slow down due to contention on disk or network bandwidth.

There are downsides to both CAC and CC. The next two sub-sections quantify these downsides.

4.2 Probability of Job Straggling: CAC vs. CC

CAC increases the vulnerability of a job to stragglers by negating the value of some of its clones. We first analytically derive the probability of a job straggling with CAC and CC, and then compare them for some representative job sizes. We use a job with n upstream and n downstream tasks, with c clones of each task.

CAC: A job straggles with CAC when either the upstream clones straggle and consequently handicap the downstream clones, or the downstream clones straggle by themselves. We start with the upstream phase first before moving to the downstream phase.

The probability that at least d upstream clones of every clone group will succeed without straggling is given by the function Ψ; p is the probability of a task straggling.

$$\Psi(n,c,d) = \text{Probability}[n \text{ upstream tasks of } c \text{ clones with } \geq d \text{ non-stragglers per clone group}]$$

$$\Psi(n,c,d) = \left(\sum_{i=0}^{c-d} \binom{c}{i} p^i (1-p)^{c-i}\right)^n \qquad (1)$$

Therefore, the probability of exactly d upstream clones not straggling is calculated as:

$$\Psi(n, c, d) - \Psi(n, c, d-1)$$

Recall that there are n downstream tasks that are cloned c times each. Therefore, the probability of the whole job straggling is essentially the probability of a straggler occurring in the downstream phase, conditional on the number of upstream clones that are non-stragglers.

$$\text{Probability[Job straggling with CAC]} = 1 - \sum_{d=1}^{c} \left[\Psi(n, c, d) - \Psi(n, c, d-1)\right] \left(1 - p^d\right)^n \quad (2)$$

CC: CC assigns all downstream clones to the output of the first upstream task that finishes in every clone group. As all the downstream clones start at the same time, none of them are handicapped. For a job to succeed without straggling, it only requires that one of the upstream clones in each clone group be a non-straggler. Therefore, the probability of the job straggling is:

$$\text{Probability[Job straggling with CC]} = 1 - \Psi(n, c, 1)\left(1 - p^c\right)^n \quad (3)$$

CAC vs. CC: We now compare the probability of a job straggling with CAC and CC for different job sizes. Figure 5 plots this for jobs with 10 and 20 upstream and downstream tasks each. With three clones per task, the probability of the job straggling increases by over 10% and 30% with CAC compared to CC. Contrast this with our algorithm in §3.2 which aims for an ϵ of 5%. The gap between CAC and CC diminishes for higher numbers of clones but this is contradictory to our decision to pick task-level cloning as we wanted to limit the number of clones. In summary, CAC significantly increases susceptibility of jobs to stragglers compared to CC.

4.3 I/O Contention with CC

By assigning all tasks in a downstream clone group to read the output of the earliest upstream clone, CC causes contention for IO bandwidth. We quantify the impact due to this contention using a micro-benchmark rather than using mathematical analysis to model IO bandwidths, which for contention is likely to be inaccurate.

With the goal of realistically measuring contention, our micro-benchmark replicates the all-to-all data shuffle portion of jobs in the Facebook trace. The experiment is performed on the same 150 node cluster we use for Dolly's evaluation (§5). Every downstream task reads its share of the output from each of the upstream tasks. All the reads start at exactly the same relative time as in the original trace and read the same amount of data from every upstream task's output. The reads of all the downstream tasks of a job together constitute a *transfer* [22].

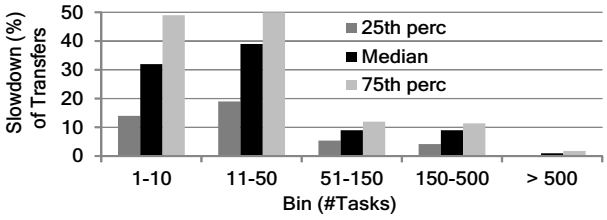


Figure 6: **Slowdown (%) of transfer of intermediate data between phases (all-to-all) due to contention by CC.**

The number of clones per upstream and downstream task is decided as in §3. In the absence of stragglers, there would be as many copies of the upstream outputs as there are downstream clones. However, a fraction of the upstream clones will be stragglers. When upstream clones straggle, we assume their copy of the intermediate data is not available for the transfer. Naturally, this causes contention among the downstream clones.

Reading contended copies of intermediate data likely results in a lower throughput than when there are exclusive copies. Of interest to us is the slowdown in the transfer of the downstream phase due to such contentions, compared to the case where there are as many copies of the intermediate data as there are downstream clones.

Figure 6 shows the slowdown of transfers in each bin of jobs. Transfers of jobs in the first two bins slow down by 32% and 39% at median, third quartile values are 50%. Transfers of large jobs are less hurt because tasks of large jobs are often not cloned because of lack of cloning budget. Overall, we see that contentions cause significant slowdown of transfers and are worth avoiding.

4.4 Delay Assignment

The analyses in §4.2 and §4.3 conclude that both CAC and CC have downsides. Contentions with CC are not small enough to be ignored. Following strict CAC is not the solution either because it diminishes the benefits of cloning. A deficiency with both CAC and CC is that they do not distinguish stragglers from tasks that have normal (but minor) variations in their progress. CC errs on the side of assuming that all clones other than the earliest are stragglers, while CAC assumes all variations are normal.

We develop a hybrid approach, *delay assignment*, that first waits to assign the early upstream clones (like CAC), and thereafter proceeds without waiting for any remaining stragglers (like CC). Every downstream clone waits for a small window of time (ω) to see if it can get an exclusive copy of the intermediate data. The wait time of ω allows for normal variations among upstream clones. If the downstream clone does not get its exclusive copy even after waiting for ω, it reads with contention from one of the finished upstream clone's outputs.

Crucial to delay assignment's performance is setting the wait time of ω. We next proceed to discuss the analysis that picks a balanced value of ω.

Setting the delay (ω)**:** The objective of the analysis is to minimize the expected duration of a downstream task, which is the minimum of the durations of its clones.

We reuse the scenario from Figure 4. After waiting for ω, the downstream clone either gets its own exclusive copy, or reads the available copy with contention with the other clone. We denote the durations for reading the data in these two cases as T_E and T_C, respectively. In estimating read durations, we eschew detailed modeling of systemic and network performance. Further, we make the simplifying assumption that all downstream clones can read the upstream output (of size r) with a bandwidth of B when there is no contention, and αB in the presence of contention ($\alpha \leq 1$).

Our analysis, then, performs the following three steps.

1. Calculate the clone's expected duration for reading each upstream output using T_C and T_E.
2. Use read durations of all clones of a task to estimate the overall duration of the task.
3. Find the delay ω that minimizes the task's duration.

Step (1)**:** We first calculate T_C, *i.e.,* the case where the clone waits for ω but does not get its exclusive copy, and contends with the other clone. The downstream clone that started reading first will complete its read in $\left(\omega + \left(\frac{r-Bw}{\alpha B}\right)\right)$, *i.e.,* it reads for ω by itself and contends with the other clone for the remaining time. The other clone takes $\left(2\omega + \left(\frac{r-Bw}{\alpha B}\right)\right)$ to read the data.

Alternately, if the clone gets its exclusive copy, then the clone that began reading first reads without interruption and completes its read in $\left(\frac{r}{B}\right)$. The other clone, since it gets its own copy too, takes $\left(\frac{r}{B} + \min(\frac{r}{B}, \omega)\right)$ to read the data.[5] Now that we have calculated T_C and T_E, the expected duration of the task for reading this upstream output is simply $p_c T_C + (1 - p_c)\, T_E$, where p_c is the probability of the task not getting an exclusive copy. Note that, regardless of the number of clones, every clone is assigned an input source latest at the end of ω. Unfinished upstream clones at that point are killed.

Step (2)**:** Every clone may have to read the outputs of multiple upstream clones, depending on the intermediate data communication pattern. In all-to-all communication, a task reads data from each upstream task's output. In one-to-one or many-to-one communications, a task reads data from just one or few tasks upstream of it. Therefore, the total time T_i taken by clone i of a task is obtained by considering its read durations from each of the relevant upstream tasks, along with the expected time for computation. The expected duration of the task is the minimum of all its clones, $\min_i (T_i)$.

Step (3)**:** The final step is to find ω that minimizes this expected task duration. We sample values of B and α, p_c and the computation times of tasks from samples of completed jobs. The value of B depends on the number of active flows traversing a machine, while the p_c is inversely proportional to ω. Using these, we pick ω that minimizes the duration of a task calculated in step (2). The value of ω is calculated periodically and automatically for different job bins (see §5.2). A subtle point with our analysis is that it automatically considers the option where clones read from the available upstream output, one after the other, without contending.

A concern in the strategy of delaying a task is that it is not work-conserving and also somewhat contradicts the observation in §2 that waiting before deciding to speculate is harmful. Both concerns are ameliorated by the fact that we eventually pick a wait duration that minimizes the completion time. Therefore, our wait is not because we lack data to make a decision but precisely because the data dictates that we wait for the duration of ω.

5 Evaluation

We evaluate Dolly using a prototype built by modifying the Hadoop framework [8]. We deploy our prototype on a 150-node cluster and evaluate it using workloads derived from the Facebook and Bing traces (§2), indicative of Hadoop and Dryad clusters. In doing so, we preserve the inter-arrival times of jobs, distribution of job sizes, and the DAG of the jobs from the original trace. The jobs in the Dryad cluster consist of multiple phases with varied communication patterns between them.

5.1 Setup

Prototype Implementation: We modify the job scheduler of Hadoop 0.20.2 [8] to implement Dolly. The two main modifications are launching clones for every task and assigning map outputs to reduce clones such that they read the intermediate data without contention.

When a job is submitted, its tasks are queued at the scheduler. For every queued task, the scheduler spawns many clones. Clones are indistinguishable and the scheduler treats every clone as if it were another task.

The all-to-all transfer of intermediate data is implemented as follows in Hadoop. When map tasks finish, they notify the scheduler about the details of their outputs. The scheduler, in turn, updates a synchronized list of available map outputs. Reduce tasks start after a fraction of the map tasks finish [23]. On startup, they poll on the synchronized list of map outputs and fetch their data as and when they become available. There are two changes we make here. First, every reduce task differen-

[5] The wait time of ω is an upper limit. The downstream clone can start as soon as the upstream output arrives.

Bin	1	2	3	4	5
Tasks	1–10	11–50	51–150	151–500	> 500

Table 3: **Job bins, binned by their number of tasks.**

tiates between map clones and avoids repetitive copying. Second, tasks in a reduce clone group notify each other when they start reading the output of a map clone. This helps them wait to avoid contention.

Deployment: We deploy our prototype on a private cluster with 150 machines. Each machine has 24GB of memory, 12 cores, and 2TB of storage. The machines have 1Gbps network links connected in a topology with full bisection bandwidth. Each experiment is repeated five times and we present the median numbers.

Baseline: Our baselines for evaluating Dolly are the state-of-the-art speculation algorithms—LATE [5] and Mantri [4]. Additionally, with each of these speculation strategies, we also include a blacklisting scheme that avoids problematic machines (as described in §2.1.2).

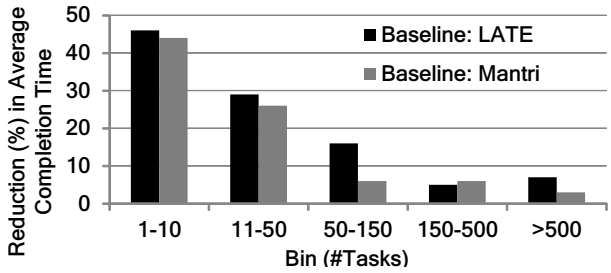
In addition to overall improvement in average completion time of jobs, we bin jobs by their number of tasks (see Table 3) and report the average improvement in each bin. The following is a summary of our results.

- Average completion time of small jobs improves by 34% to 46% compared to LATE and Mantri, using fewer than 5% extra resources (§5.2 and §5.4).
- Delay assignment outperforms CAC and CC by 2×. Its benefit increases for jobs with higher number of phases and all-to-all intermediate data flow (§5.3).
- Admission control of jobs is a good approximation for preemption in favoring small jobs (§5.5).

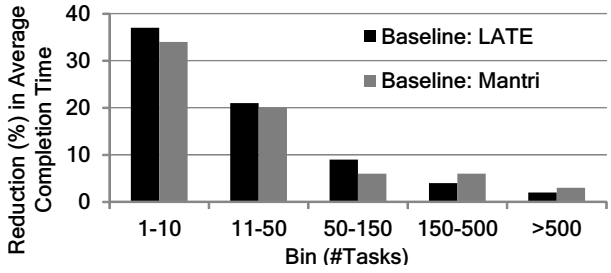
5.2 Does Dolly mitigate stragglers?

We first present the improvement in completion time using Dolly. Unless specified otherwise, the cloning budget β is 5% and utilization threshold τ is 80%.

Dolly improves the average completion time of jobs by 42% compared to LATE and 40% compared to Mantri, in the Facebook workload. The corresponding improvements are 27% and 23% in the Bing workload. Figure 7 plots the improvement in different job bins. Small jobs (bin-1) benefit the most, improving by 46% and 37% compared to LATE and 44% and 34% compared to Mantri, in the Facebook and Bing workloads. This is because of the power-law in job sizes and the policy of admission control. Figures 8a and 8b show the average duration of jobs in the smallest two bins with LATE and Mantri, and its reduction due to Dolly's cloning, for the Facebook workload. Figure 8c shows the distribution of gains for jobs in bin-1. We see that jobs improve by



(a) Facebook workload.



(b) Bing workload.

Figure 7: Dolly**'s improvement for the Facebook and Bing workloads, with LATE and Mantri as baselines.**

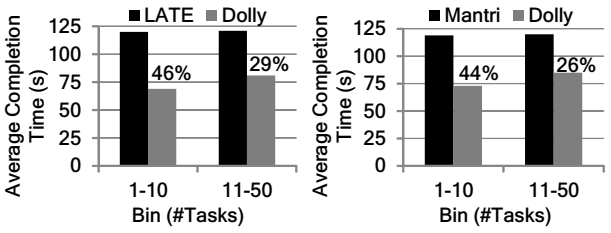
nearly 50% and 60% at the 75$^{\text{th}}$ and 90$^{\text{th}}$ percentiles, respectively. Note that even at the 10$^{\text{th}}$ percentile, there is a non-zero improvement, demonstrating the seriousness and prevalence of the problem of stragglers in small jobs.

Figure 9 presents supporting evidence for the improvements. The ratio of medium to minimum progress rates of tasks, which is over 5 with LATE and Mantri in our deployment, drops to as low as 1.06 with Dolly. Even at the 95$^{\text{th}}$ percentile, this ratio is only 1.17, thereby indicating that Dolly effectively mitigates nearly all stragglers.

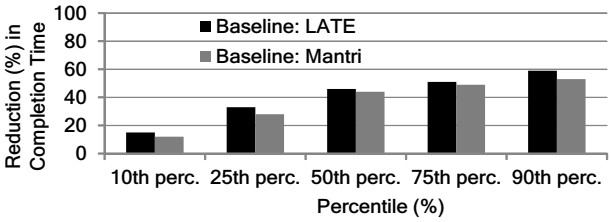
The ratio not being exactly 1 shows that some stragglers still remain. One reason for this is that while our policy of admission control is a good approximation (§3.3), it does not explicitly prioritize small jobs. Hence a few large jobs possibly deny the budget to some small jobs. Analyzing the consumption of the cloning budget shows that this is indeed the case. Jobs in bin-1 and bin-2 together consume 83% of the cloning budget. However, even jobs in bin-5 get a small share (2%) of the budget.

5.3 Delay Assignment

Setting ω: Crucial to the above improvements is delay assignment's dynamic calculation of the wait duration of ω. The value of ω, picked using the analysis in §4.4, is updated every hour. It varied between 2.5s and 4.7s for jobs in bin-1, and 3.1s and 5.2s for jobs in bin-2. The value of ω varies based on job sizes because the number of tasks in a job influences B, α and p_c. Figure 10 plots the variation with time. The sensitivity of ω to the periodicity of updating its value is low—using values between

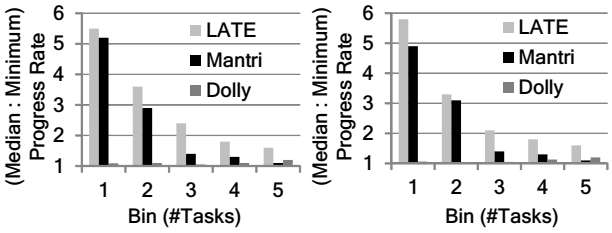


(a) Job Durations. (b) Job Durations.



(c) Distribution of improvements ($\leq$ 10 tasks).

Figure 8: **Dissecting Dolly's improvements for the Facebook workload. Figures (a) and (b) show the duration of the small jobs before and after Dolly. Figure (c) expands on the distribution of the gains for jobs with $\leq$ 10 tasks.**



(a) Facebook (b) Bing

Figure 9: **Ratio of median to minimum progress rates of tasks within a phase. Bins are as per Table 3.**

30 minutes to 3 hours causes little change in its value.

CC and CAC: We now compare delay assignment to the two static assignment schemes, Contention Cloning (CC) and Contention Avoidance Cloning (CAC) in Figure 11, for the Bing workload. With LATE as the baseline, CAC and CC improve the small jobs by 17% and 26%, in contrast to delay assignment's 37% improvement (or up to 2.1× better). With Mantri as the baseline, delay assignment is again up to 2.1× better. In the Facebook workload, delay assignment is at least 1.7× better.

The main reason behind delay assignment's better performance is its accurate estimation of the effect of contention and the likelihood of stragglers. It uses sampling from prior runs to estimate both. Bandwidth estimation is 93% accurate without contention and 97% accurate with contention. Also, the probability of an upstream

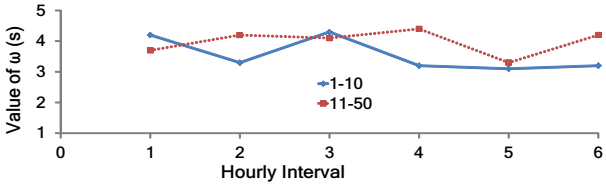
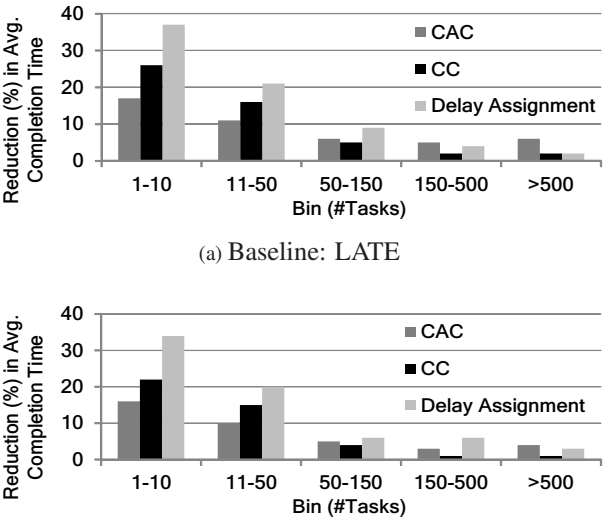


Figure 10: **Variation in ω when updated every hour.**



(a) Baseline: LATE

(b) Baseline: Mantri

Figure 11: **Intermediate data contention. Delay Assignment is 2.1× better than CAC and CC (Bing workload).**

clone straggling is estimated to an accuracy of 95%.

Between the two, CC is a closer competitor to delay assignment than CAC, for small jobs. This is because they transfer only moderate amounts of data. However, contentions hurt large jobs as they transfer sizable intermediate data. As a result, CC's gains drop below CAC.

Number of Phases: Dryad jobs may have multiple phases (maximum of 6 in our Bing traces), and tasks of different phases have the same number of clones. More phases increases the chances of there being fewer exclusive copies of task outputs, which in turn worsens the effect of both waiting as well as contention. Figure 12 measures the consequent drop in performance. CAC's gains drop quickly while CC's performance drops at a moderate rate. Importantly, delay assignment's performance only has a gradual and relatively small drop. Even when the job has six phases, improvement is at 31%, a direct result of its deft cost-benefit analysis (§4.4).

Communication Pattern: Delay assignment is generic to handle any communication pattern between phases. Figure 13 differentiates the gains in completion times of the *phases* based on their communication pattern. Results show that delay assignment is significantly more

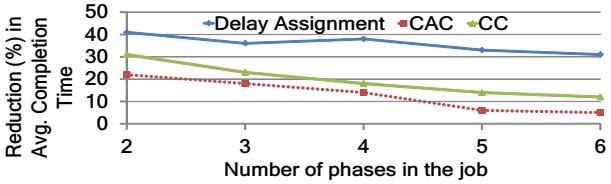


Figure 12: Dolly's gains as the number of phases in jobs in bin-1 varies in the Bing workload, with LATE as baseline.

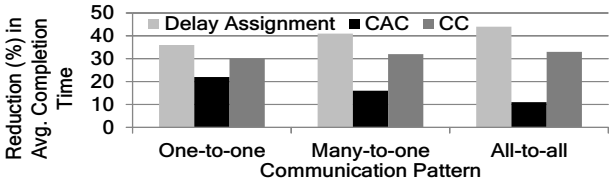
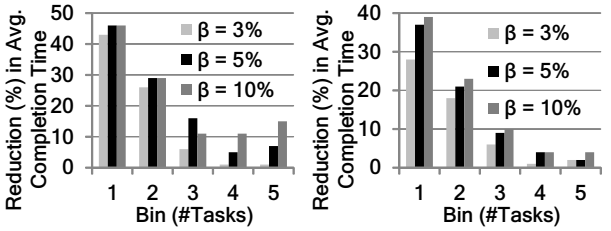


Figure 13: Performance of Dolly across phases with different communication patterns in bin-1, in the Bing workload.



(a) Facebook (b) Bing

Figure 14: Sensitivity to cloning budget (β). Small jobs see a negligible drop in performance even with a 3% budget.

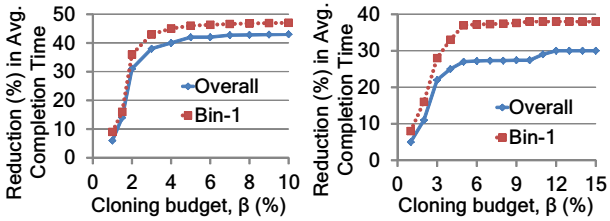
valuable for all-to-all communication patterns than the many-to-one and one-to-one patterns. The higher the dependency among communicating tasks, the greater the value of delay assignment's cost-benefit analysis.

Overall, we believe the above analysis shows the applicability and robust performance of Dolly's mechanisms to different frameworks with varied features.

5.4 Cloning Budget

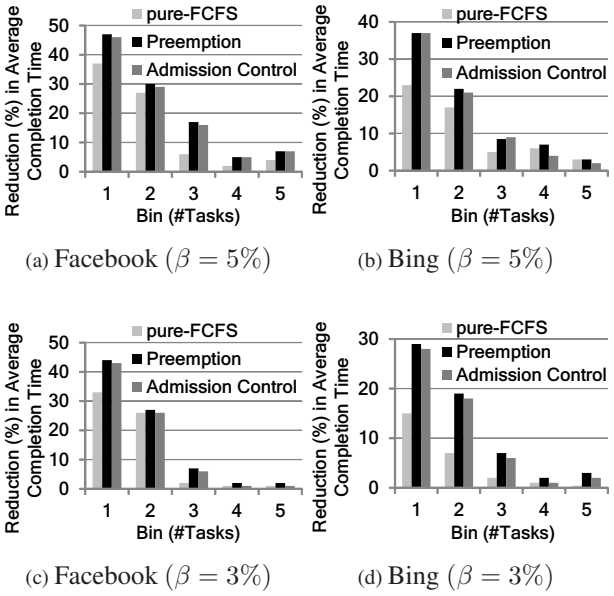
The improvements in the previous sections are based on a cloning budget β of 5%. In this section, we analyze the sensitivity of Dolly's performance to β. We aim to understand whether the gains hold for lower budgets and how much further gains are obtained at higher budgets.

In the Facebook workload, overall improvement remains at 38% compared to LATE even with a cloning budget of only 3% (Figure 14a). Small jobs, in fact, see a negligible drop in gains. This is due to the policy of admission control to favor small jobs. Large jobs take a non-negligible performance hit though. In fact, in the Bing workload, even the small jobs see a drop of 7%



(a) Facebook (b) Bing

Figure 15: Sweep of β to measure the overall average completion time of all jobs and specifically those within bin-1.



(a) Facebook ($\beta = 5\%$) (b) Bing ($\beta = 5\%$)

(c) Facebook ($\beta = 3\%$) (d) Bing ($\beta = 3\%$)

Figure 16: **Admission Control. The policy of admission control well approximates the policy of preemption and outperforms pure-FCFS in utilizing the cloning budget.**

when the budget is reduced from 5% to 3%. This is because job sizes in Bing are less heavy-tailed. However, the gains still stand at a significant 28% (Figure 14b).

Increasing the budget to 10% does not help much. Most of the gains are obtained by eliminating stragglers in the smaller jobs, which do not require a big budget.

In fact, sweeping the space of β (Figure 15) reveals that Dolly requires a cloning budget of at least 2% and 3% for the Facebook and Bing workloads, below which performance drops drastically. Gains in the Facebook workload plateau beyond 5%. In the Bing workload, gains for jobs in bin-1 plateau at 5% but the overall gains cease to grow only at 12%. While this validates our setting of β as 5%, clusters can set their budgets based on their utilizations and the jobs they seek to improve with cloning.

5.5 Admission Control

A competing policy to admission control (§3.3) is to preempt clones of larger jobs for the small jobs. Preemption is expected to outperform admission control as it explicitly prioritizes the small jobs; we aim to quantify the gap.

Figure 16 presents the results with LATE as the baseline and cloning budgets of 5% and 3%. The gains with preemption is 43% and 29% in the Facebook and Bing workloads, compared to 42% and 27% with the policy of admission control. This small difference is obtained by preempting 8% and 9% of the tasks in the two workloads. Lowering the cloning budget to 3% further shrinks this difference, even as more tasks are preempted. With a cloning budget of 3%, the improvements are nearly equal, even as 17% of the tasks are preempted, effectively wasting cluster resources. Admission control well approximates preemption due to the heavy tailed distribution. Note the near-identical gains for small jobs.

Doing neither preemption or admission control in allocating the cloning budget ("pure-FCFS") reduces the gains by nearly 14%, implying this often results in larger jobs denying the cloning budget to the smaller jobs.

6 Related Work

Replicating tasks in distributed systems have a long history [24, 25, 26], and have been studied extensively [27, 28, 29] in prior work. These studies conclude that modeling running tasks and using it for predicting and comparing performance of other tasks is the hardest component, errors in which often cause degradation in performance. We concur with a similar observation in our traces.

The problem of stragglers was identified in the original MapReduce paper [1]. Since then solutions have been proposed to fix it using speculative executions [2, 4, 5]. Despite these techniques, stragglers remain a problem in small jobs. Dolly addresses their fundamental limitation—wait to observe before acting—with a proactive approach of cloning jobs. It does so using few extra resources by relying on the power-law of job sizes.

Based on extensive research on detecting faults in machines (*e.g.,* [30, 31, 32, 33, 34]), datacenters periodically check for faulty machines and avoid scheduling jobs on them. However, stragglers continue to occur on the non-blacklisted machines. Further improvements to blacklisting requires a root cause analysis of stragglers in small jobs. However, this is intrinsically hard due to the complexity of the hardware and software modules, a problem recently acknowledged in Google's clusters [6].

In fact, Google's clusters aim to make jobs "predictable out of unpredictable parts" [6]. They overcome vagaries in performance by scheduling backup copies for every job. Such backup requests are also used in Amazon's Dynamo [35]. This notion is similar to Dolly. However, these systems aim to overcome variations in scheduling delays on the machines, not runtime stragglers. Therefore, they cancel the backup copies once one of the copies starts. In contrast, Dolly has to be resilient to runtime variabilities which requires functioning within utilization limits and efficiently handle intermediate data.

Finally, our delay assignment model is similar to the idea of delay scheduling [36] that delays scheduling tasks for locality. We borrow this idea in Dolly, but crucially, pick the value of the delay based on a cost-benefit analysis weighing contention versus waiting for slower tasks.

7 Conclusions and Future Work

Analysis of production traces from Facebook and Microsoft Bing show that straggler tasks continue to affect small interactive jobs by 47% even after applying state-of-the-art mitigation techniques [4, 5]. This is because these techniques wait before launching speculative copies. Such waiting bounds their agility for small jobs that run all their tasks at once.

In this paper we developed a system, Dolly, that launches multiple clones of jobs, completely removing waiting from straggler mitigation. Cloning of small jobs can be achieved with few extra resources because of the heavy-tail distribution of job sizes; the majority of the jobs are small and can be cloned with little overhead. The main challenge of cloning was making the intermediate data transfer efficient, *i.e.,* avoiding multiple tasks downstream in the job from contending for the same upstream output. We developed *delay assignment* to efficiently avoid such contention using a cost-benefit model. Evaluation using production workloads showed that Dolly sped up small jobs by 34% to 46% on average, after applying LATE and Mantri, using only 5% extra resources.

Going forward, we plan to evaluate Dolly's compatibility with caching systems proposed for computation frameworks. These systems rely on achieving *memory locality*—scheduling a task on the machine that caches its input—along with cache replacement schemes targeted for parallel jobs [37]. Analyzing (and dealing with) the impact of multiple clones for every task on both these aspects is a topic for investigation.

We also plan to extent Dolly to deal with clusters that deploy multiple computation frameworks. Trends indicate a proliferation of frameworks, based on different computational needs and programming paradigms (*e.g.,* [3, 7]). Such specialized frameworks may, perhaps, lead to homogeneity of job sizes within them. Challenges in extending Dolly to such multi-framework clusters includes dealing with any weakening of the heavy-tail distribution, a crucial factor behind Dolly's low overheads.

References

[1] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *USENIX OSDI*, 2004.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *ACM Eurosys*, 2007.

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *USENIX NSDI*, 2012.

[4] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, E. Harris, and B. Saha. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *USENIX OSDI*, 2010.

[5] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *USENIX OSDI*, 2008.

[6] J. Dean. *Achieving Rapid Response Times in Large Online Services*. `http://research.google.com/people/jeff/latency.html`.

[7] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. In *VLDB*, 2010.

[8] Hadoop. `http://hadoop.apache.org`.

[9] Hadoop distributed file system. `http://hadoop.apache.org/hdfs`.

[10] Hive. `http://wiki.apache.org/hadoop/Hive`.

[11] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB*, 2008.

[12] Y. Yu *et al.* Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations. In *ACM SOSP*, 2009.

[13] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True Elasticity in Multi-Tenant Clusters through Amoeba. In *ACM SoCC*, 2012.

[14] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. A Study of Skew in MapReduce Applications. In *Open Cirrus Summit*, 2011.

[15] L. A. Barroso. Warehouse-scale computing: Entering the teenage decade. In *ISCA*, 2011.

[16] Y. Chen, S. Alspaugh, D. Borthakur, R. Katz. Energy Efficiency for Large-Scale MapReduce Workloads with Significant Interactive Analysis. In *ACM EuroSys*, 2012.

[17] J. Wilkes and C. Reiss., 2011. `https://code.google.com/p/googleclusterdata/wiki/ClusterData2011_1`.

[18] C. Reiss, A. Tumanov, G. Ganger, R. H. Katz, M. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM SoCC*, 2012.

[19] A. Thusoo . Data warehousing and analytics infrastructure at facebook. In *SIGMOD*, 2010.

[20] G. Ananthanarayanan, A. Ghodsi, S. Shenker, I. Stoica. Disk Locality Considered Irrelevant. In *USENIX HotOS*, 2011.

[21] S. Ko, I. Hoque, B. Cho, I. Gupta. Making Cloud Intermediate Data Fault-Tolerant. In *ACM SOCC*, 2010.

[22] M. Chowdhury, M. Zaharia, J. Ma, M. Jordan, I. Stoica. Managing Data Transfers in Computer Clusters with Orchestra. In *ACM SIGCOMM*, 2011.

[23] Hadoop Slowstart. `https://issues.apache.org/jira/browse/MAPREDUCE-1184/`.

[24] A. Baratloo, M. Karaul, Z. Kedem, and P. Wycko. Charlotte: Metacomputing on the Web. In *9th Conference on Parallel and Distributed Computing Systems*, 1996.

[25] E. Korpela D. Anderson, J. Cobb. SETI@home: An Experiment in Public-Resource Computing. In *Comm. ACM*, 2002.

[26] M. C. Rinard and P. C. Diniz. Commutativity Analysis: A New Analysis Framework for Parallelizing Compilers. In *ACM PLDI*, 1996.

[27] D. Paranhos, W. Cirne, and F. Brasileiro. Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids. In *Euro-Par*, 2003.

[28] G. Ghare and S. Leutenegger. Improving Speedup and Response Times by Replicating Parallel Programs on a SNOW. In *JSSPP*, 2004.

[29] W. Cirne, D. Paranhos, F. Brasileiro, L. F. W. Goes, and W. Voorsluys. On the Efficacy, Efficiency and Emergent Behavior of Task Replication in Large Distributed Systems. In *Parallel Computing*, 2007.

[30] A. Merchant, M. Uysal, P. Padala, X. Zhu, S. Singhal, and K. Shin. Maestro: Quality-of-Service in Large Disk Arrays. In *ACM ICAC*, 2011.

[31] E. Ipek, M. Krman, N. Krman, and J. F. Martinez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *ISCA*, 2007.

[32] J. G. Elerath and S. Shah. Dependence upon fly-height and quantity of heads. In *Annual Symposium on Reliability and Maintainability*, 2003.

[33] J. G. Elerath and S. Shah. Server class disk drives: How reliable are they? In *Annual Symposium on Reliability and Maintainability*, 2004.

[34] J. Gray and C. van Ingen. Empirical measurements of disk failure rates and error rates. In *Technical Report MSR-TR-2005-166*, 2005.

[35] G. DeCandia and D. Hastorun and M. Jampani and G. Kakulapati and A. Lakshman and A. Pilchin and S. Sivasubramanian and P. Vosshall and W. Vogels. Dynamo: Amazons Highly Available Key-value Store. In *ACM SOSP*, 2007.

[36] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *ACM EuroSys*, 2010.

[37] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *USENIX NSDI*, 2012.

Wire Speed Name Lookup: A GPU-based Approach

Yi Wang[†], Yuan Zu[‡], Ting Zhang[†], Kunyang Peng[‡], Qunfeng Dong[‡©], Bin Liu[†©],
Wei Meng[†], Huichen Dai[†], Xin Tian[‡], Zhonghu Xu[‡], Hao Wu[†], Di Yang[‡]
[†]*Tsinghua National Laboratory for Information Science and Technology,*
Department of Computer Science and Technology, Tsinghua University
[‡]*Institute of Networked Systems (IONS) & School of Computer Science and Technology,*
University of Science and Technology of China

Abstract

This paper studies the name lookup issue with longest prefix matching, which is widely used in URL filtering, content routing/switching, etc. Recently Content-Centric Networking (CCN) has been proposed as a clean slate future Internet architecture to naturally fit the content-centric property of today's Internet usage: instead of addressing end hosts, the Internet should operate based on the identity/name of contents. A core challenge and enabling technique in implementing CCN is exactly to perform name lookup for packet forwarding at wire speed. In CCN, routing tables can be orders of magnitude larger than current IP routing tables, and content names are much longer and more complex than IP addresses. In pursuit of conquering this challenge, we conduct an implementation-based case study on wire speed name lookup, exploiting GPU's massive parallel processing power. Extensive experiments demonstrate that our GPU-based name lookup engine can achieve 63.52M searches per second lookup throughput on large-scale name tables containing millions of name entries with a strict constraint of no more than the telecommunication level 100μs per-packet lookup latency. Our solution can be applied to contexts beyond CCN, such as search engines, content filtering, and intrusion prevention/detection.

©Prof. Qunfeng Dong (qunfeng.dong@gmail.com) and Prof. Bin Liu (lmyujie@gmail.com), placed in alphabetic order, are the correspondence authors of the paper. Yi Wang and Yuan Zu, placed in alphabetic order, are the lead student authors of Tsinghua University and University of Science and Technology of China, respectively.

This paper is supported by 863 project (2013AA013502), NSFC (61073171, 61073184), Tsinghua University Initiative Scientific Research Program(20121080068), the Specialized Research Fund for the Doctoral Program of Higher Education of China(20100002110051), the Ministry of Education (MOE) Program for New Century Excellent Talents (NCET) in University, the Science and Technological Fund of Anhui Province for Outstanding Youth (10040606Y05), by the Fundamental Research Funds for the Central Universities (WK0110000007, WK0110000019), and Jiangsu Provincial Science Foundation (BK2011360).

1 Introduction

Name lookup is widely used in a broad range of technological fields, such as search engine, data center, storage system, information retrieval, database, text processing, web application, programming languages, intrusion detection/prevention, malware detection, content filtering and so on. Most of these name lookup applications either perform exact matching only or operate on small-scale data sets. The recently emerging Content-Centric Networking (CCN) [12] proposes to use a content name to identify a piece of data instead of using an IP address to locate a device. In CCN scenario, every distinct content/entity is referenced by a unique name. Accordingly, communication in CCN is no longer address-based, but name-based. CCN routers forward packets based on the requested content name(s) carried in each packet header, by looking up a forwarding table consisting of content name prefixes.

CCN name lookup complies with longest prefix matching (LPM) and backbone CCN routers can have large-scale forwarding tables. Wire speed name lookup presents a research challenge because of stringent requirements on memory occupation, throughput, latency and fast incremental update. Practical name lookup engine design and implementation, therefore, require elaborate design-level innovation plus implementation-level re-engineering.

1.1 Names and Name Tables

Content naming, as recently proposed in the Named Data Network (NDN) project [28][1], is hierarchically structured and composed of explicitly delimited name components, such as reversed domain names followed by directory-style path. For in-

[1]CCN refers to the general content-centric networking paradigm; NDN refers to the specific proposal of the NDN project. However, we shall use them interchangeably in the rest of the paper.

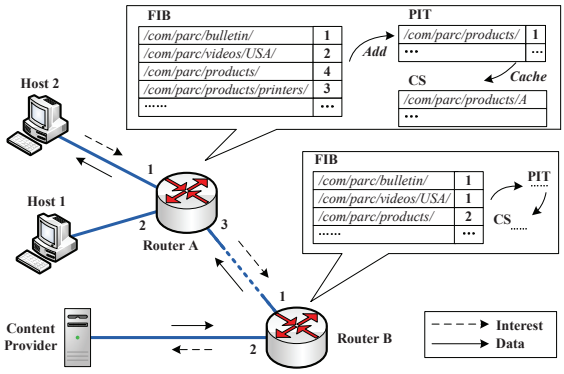


Figure 1: NDN communication example[3].

stance, `com/parc/bulletin/NSDI.html` is an example NDN content name, where `com/parc/` is the reversed domain name `parc.com` of the web site and `/bulletin/NSDI.html` is the content's directory path on the web site. '`/`' is the component boundary delimiter and not a part of the name; `com`, `parc`, `bulletin` and `NSDI.html` are four components of the name.

The format of *Forwarding Information Base (FIB)*[2] of NDN routers is shown beside Router A and B in Figure 1. Each FIB entry is composed of a name prefix and the corresponding outgoing port(s). NDN name lookup also complies with *longest prefix matching (LPM)*. For example, suppose the content name is `/com/parc/products/printers/hp`, which matches the third and fourth entries in Router A; the fourth entry as the longest matching prefix determines that the packet should be forwarded through port 3.

1.2 Challenges

To implement CCN routing with large-scale FIB tables in high speed networks, a core challenge and enabling technique is to perform content name lookup for forwarding packets at wire speed. In particular, a name lookup engine is confronted with the following difficulties.

First, content names are far more complex than IP addresses. As introduced above, content names are much longer than IPv4/IPv6 addresses; each name is composed of tens, or even hundreds, of characters. In addition, unlike fixed-length IP addresses, content names have variable lengths, which further complicates the name lookup.

Second, CCN name tables could be far larger than today's IP forwarding tables. Compared with the current IP routing tables with up to 400K IP prefix entries, CCN name tables could be orders of magnitude larger. Without elaborate compression and implementation, they can by far exceed the capacity of today's commodity devices.

Third, wire speeds have been relentlessly accelerating. Today, OC-768 (40Gbps) links have already been deployed in Internet backbone, and OC-3072 (160Gbps) technology is emerging at the horizon of Internet.

Fourth, in addition to network topology changes and routing policy modifications, CCN routers have to handle one new type of FIB update — when contents are published/deleted, name prefixes may need to be inserted into or deleted from FIBs. This makes FIB update much more frequent than in today's Internet. Fast FIB update, therefore, must be well handled for large-scale FIBs.

1.3 Our work

In pursuit of conquering these challenges, we conduct an implementation-based case study of wire speed name lookup in large-scale name tables, exploiting GPU's massive parallel processing power.

1) We present the first design, implementation and evaluation of a GPU-based name lookup engine. Through this implementation-based experimental study, we demonstrate the feasibility of implementing wire speed name lookup with large-scale name tables at low cost, using today's commodity GPU devices. We have released the implementation code, data traces and documents of our work [3].

2) Our GPU-based name lookup engine is featured by a new technique called *multiple aligned transition arrays (MATA)*, which combines the best of two worlds. On one hand, MATA effectively improves lookup speed by reducing the number of memory access. On the other hand, MATA as one-dimensional arrays can substantially compress storage space. Due to these unique merits, MATA is demonstrated through experiments to be able to compress storage space by two orders of magnitude, while promoting lookup speed by an order of magnitude, compared with two-dimensional state transition tables.

3) GPU achieves high processing throughput by exploiting massive data-level parallelism — large amounts of input data (i.e., names) are loaded into GPU, looked up in GPU and output results from GPU together to hide GPU's DRAM access latency. While effectively boosting processing throughput, this typical GPU design philosophy easily leads to extended per-packet latency. In this work, we take on this throughput-latency dilemma by exploiting the *multi-stream* mechanism featured in NVIDIA's Fermi GPU family. Our stream-based pipeline solution ensures practical per-packet latency (less than 100μs) while keeping high lookup throughput.

4) We employ data interweaving [32] technique for optimizing the storage of input names in GPU memory.

[2] In this paper, we shall use three terms — *FIB*, *FIB table* and *name table* — interchangeably.

[3] In the NDN proposal, there are three kinds of tables, FIB, PIT and CS. Only if the CS and PIT both fail to match, name lookup in FIB is performed. When we evaluate our lookup engine, we assume this worst case where every name has to be looked up in FIB.

As a result, memory access efficiency is significantly improved, further boosting name lookup performance.

We implement our name lookup engine on a commodity PC installed with an NVIDIA GeForce GTX590 GPU board. Using real world URL names collected from Internet, we conduct extensive experiments to evaluate and analyze the performance of our GPU-based name lookup engine. On large-scale name tables containing up to 10M names, the CPU-GPU lookup engine obtains 63.52M searches per second (SPS) under average workload, enabling an average line rate of 127 Gbps (with 256-byte average packet size). Even under heavy workload, we can still obtain up to 55.65 MSPS, translating to 111 Gbps wire speed. Meanwhile, lookup latency can be as low as 100μs. In fact, if the PCIe bus bandwidth between CPU and GPU were not the system bottleneck, the lookup engine core running on the GPU could achieve 219.69 MSPS! Besides, experiments also show that our name lookup engine can support fast incremental name table update.

These results advocate our GPU-based name lookup engine design as a practical solution for wire speed name lookup, using today's off-the-shelf technologies. The results obtained in this work, however, will have broad impact on many technological fields other than CCN.

2 Algorithms & Data Structures

In this section, we present the core algorithmic and data structure design of our GPU-based name lookup engine. The entire design starts with name table aggregation in Section 2.1, where name tables are aggregated into smaller yet equivalent ones. After that, we present in Section 2.2 aligned transition array (ATA), which subsequently evolves into multi-striding ATA in Section 2.3 and multi-ATA (MATA) — the core data structure for high speed and memory efficient name lookup — in Section 2.4. Finally in Section 2.5, we demonstrate how name table updates can be handled with ease in our lookup engine design.

2.1 Name table aggregation

The hierarchical structure of NDN names and the longest prefix matching property of NDN name lookup enable us to aggregate NDN name tables into smaller ones. For example, consider Router A's name table in Figure 1. If the third entry and the fourth entry map to the same next hop port, they can be aggregated into one, by removing the fourth entry. After this aggregation, names originally matching the fourth entry will now match the third one. Since the two entries are hereby assumed to map to the same port, it is safe to perform this aggregation.

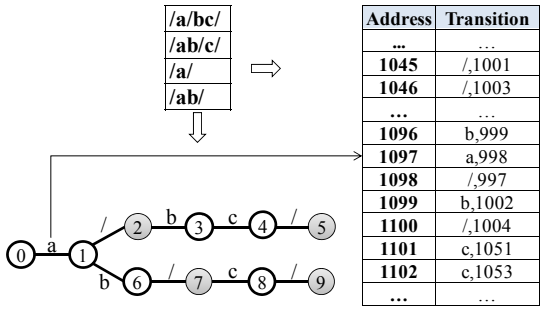


Figure 2: Aligned transition array.

To safely aggregate the two name table entries, they need to comply with two simple principles: (1) One of them is the shortest prefix of the other in the name table; (2) They must map to the same next hop port(s).

2.2 Aligned transition array (ATA)

A natural approach to implementing NDN name lookup is to build a character-trie [9], which is essentially a finite state machine (FSM) for matching incoming names against a name table, as shown in the left part of Figure 2. Each node in the character-trie is implemented as a state in the FSM, and transitions to its child nodes (i.e., states) on specified input characters. To start with, we assume the FSM processes one input character on each state transition. In such a 1-stride FSM, each state has 256 transitions, and each transition corresponds to a distinct input character. The entire FSM is essentially a two-dimensional state transition table, where the ith row implements the ith state s_i and the jth column corresponds to the jth character c_j in the input alphabet Σ, which in this case is the ASCII character set. The table entry at the intersection of the ith row and jth column records the destination state we should transit to, if the current state is s_i and the input character is c_j; an empty entry indicates failing to match. Using current state ID as row number and input character as column number, one memory access is sufficient to perform every state transition.

However, this standard solution does not work in practice. To see that, we experiment with a real name table consisting of 2,763,780 names (referred as "3M name table" for brevity). After aggregation, the constructed 1-stride FSM consists of 20,440,366 states; four bytes are needed for encoding state ID, and $256 \times 4 = 1,024$ bytes are thus needed for each row of the state transition table. The entire state transition table takes 19.49 GB memory space. In fact, the largest name table used in our experiments is even several times larger than the 3M name table. To fit such a large-scale name table into commodity GPU devices, the FSM has to be compressed by at least 2-3 orders of magnitude, while still has to perform name

lookup at wire speed, meeting stringent lookup latency requirement and supporting fast incremental name table update. This is a key challenge in the design and implementation of a practical name lookup engine, which we take on in this work.

As we can observe from Figure 2, the FSM for name lookup demonstrates a key feature — most states have valid transitions on very few input characters. For example in Figure 2, state 0 only has a valid transition on character `a`. This intuitive observation is also verified through experiments. For example, in the 1-stride FSM constructed from the 3M name table, more than 80% of states have only one single valid transition, plus more than 13% of states (which are accepting states) that have no valid transition at all. The state transition table is thus a rather sparse one.

In light of this observation, we store valid transitions into what we call an *aligned transition array (ATA)*. The basic idea is to take the sum of current state ID and input character as an index into the transition array. For example, if the current state ID is 1,000 and the input character is `a` (whose ASCII code is 97), we shall take the transition stored in the 1,097th transition array element as our valid transition[4]. To properly implement, we need to assign each state s a unique state ID such that no two valid transitions are mapped to the same transition array element. For that, we first find the smallest input character on which state s has a valid transition; suppose the character is the kth character in input alphabet Σ. Then, we find the lowest vacant element in the transition array, and suppose it is the ℓth element in the transition array. The number ℓ-k, if previously unused as a state ID, is considered as a candidate state ID for state s. To avoid possible storage collision, we need to check every input character c on which state s has a valid transition. If no collision is detected on any valid transition of state s, ℓ-k is assigned as the state ID of state s. Otherwise, if the (ℓ-k+c)th transition array element is already occupied, a collision is detected and ℓ-k is not good as the state ID for state s. The next vacant elements in the transition array are probed one by one until finding the available element.

Another mistake that can potentially happen here is that, even if current state s has not a valid transition on the current input character, (state ID + input character) may mistakenly refer to a stored valid transition belonging to another state. To handle this problem, we store for each valid transition not only its destination state ID, but also its input character for verification.

With this aligned transition array design, the example

[4]This basic idea of storing a sparse table as a one-dimensional array is introduced by Robert Endre Tarjan and Andrew Chi-Chih Yao [23] back in 1979. With our new techniques proposed in subsequent sections, we shall be able to effectively boost name lookup speed while further reducing storage space.

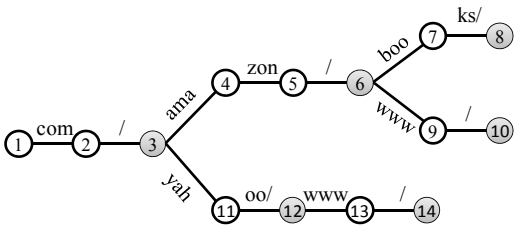


Figure 3: A 3-stride FSM.

name table in the left part of Figure 2 is implemented as the aligned transition array in the right part of Figure 2. State transition is as efficient as using two-dimensional state transition table. We simply take the sum of current state ID and input character, and read the transition array element indexed by the sum. If the stored character is the same as input character, the obtained transition directly gives us the destination state; otherwise, failing to match is detected. One single memory access is still sufficient for every state transition. Meanwhile, only valid transitions need to be stored, leading to significant compression of storage space.

Thus far, this basic ATA design looks perfect. However, scale can completely change the nature of problems. Given the limited resource and capacity of today's commodity devices, as we shall observe in subsequent sections, performing name lookup at 10Gbps in a name table containing hundreds of thousands name prefixes is one thing, while performing name lookup at 100Gbps in a name table consisting of millions of name prefixes is totally different. Especially, we also have to meet stringent per-packet lookup latency requirement and support fast incremental name table update. In subsequent sections, we shall unfold and solve various technical issues, as we proceed towards wire speed name lookup with large-scale name tables.

2.3 Multi-striding

The storage efficiency and lookup speed of aligned transition array can be further improved with multi-striding — instead of processing one character per state transition, d characters are processed on each state transition. The same algorithm for constructing 1-stride FSMs can be used to construct multi-stride FSMs. To ensure proper matching of name prefixes and to facilitate name table update, component delimiter '`/`' can only be the last character we read upon each state transition. Thus, the d-stride FSM we construct is actually an FSM of variable strides, which processes *up to* d characters ($8d$ bits) per state transition. For example, Figure 3 shows the 3-stride FSM constructed from the following three names: `/com/amazon/books/`, `/com/amazon/www/` and `/com/yahoo/www/`. Upon state transition, we keep reading in d input characters unless '`/`' is encountered,

where we stop. These input characters are transformed into an integer, which is taken as the *input number*.

By processing multiple bytes per state transition, multi-striding effectively accelerates name lookup. Even better, multi-striding also helps reduce storage space. Because a large number of intermediate states and transitions in the 1-stride FSM will be consolidated.

2.4 Multi-ATA (MATA)

While multi-striding is very effective on boosting lookup speed and reducing storage space, trying to further expand its power using larger strides leads us to an inherent constraint of the basic ATA design — its multi-striding is limited by available memory. For example, in the 3-stride FSM in Figure 3, it takes two state transitions to match the first-level name component `'com/'`, making us naturally think about using a 4-stride FSM so that one state transition will be enough. Unfortunately, in a d-stride FSM, a state can have 2^{8d} transitions at most, so the distance between a state's two valid transitions stored in the aligned transition array can be as large as 2^{8d}-1. Thus, with the 3GB memory space available on the NVIDIA GTX590 GPU board used in our experiments, 3-stride has been the maximum stride that can be implemented with basic ATA; 4-stride is not a practical option.

We break this constraint of basic ATA by defining a maximum ATA length L ($L < 2^{8d}$). For a state with state ID x, its valid transition on input number y can be stored in the ((x+y) mod L)th transition array element instead of the (x+y)th element. However, suppose state x has another valid transition on input number z. If y-z is a multiple of L, the two valid transitions will be mapped to the same transition array element and hence cause a storage collision.

For solving the above problem, we shall use a set of prime numbers L_1, L_2, ..., L_k, such that $L_1 \times L_2 \times \ldots \times L_k \geq 2^{8d}$. Accordingly, instead of creating one huge ATA, we create a number of small ATAs, each ATA using one of the prime numbers as its maximum length. Then, we first try to store the two valid transitions on y and z into an ATA with prime number L_1. (There can be multiple ATAs having the same maximum length.) If the two valid transitions do not collide with each other but collide with some valid transition(s) previously stored in that ATA, we shall try another ATA with the same maximum length; if the two valid transitions collide with each other, we shall move on trying to store state x into an ATA with a different maximum length, until ATAs with all different maximum lengths have been tried. It is guaranteed that, there must be at least one prime number L_i that can be used to store the two valid transitions without any collision. To prove by contradiction, assume the two valid transitions collide with all prime numbers L_1, L_2, ..., L_k as the maximum length. That means, y-z is a multiple of all these prime numbers L_1, L_2, ..., L_k, and hence a multiple of $L_1 \times L_2 \times \ldots \times L_k$; this in turn means $y\text{-}z \geq L_1 \times L_2 \times \ldots \times L_k \geq 2^{8d}$, which is impossible.

For each state in the FSM, as part of its state ID information, we record the small ATA that is used to store its valid transition(s). Thus, one single memory access is still sufficient to perform every state transition.

To handle cases where a state has multiple pairs of valid transitions colliding with each other, the above design can be simply augmented with more prime numbers.

In addition to breaking the constraint on maximum stride, the above described multi-ATA (MATA) design also has two other merits.

First, ATAs can leave elements unused in vacancy, due to storage collision or insufficient number of valid transitions to store. By defining maximum ATA length, we now have better control over the amount of ATA elements that are wasted in vacancy.

Second, constructing the MATA optimally is NP-hard. One single large basic ATA can take prohibitively long time to construct, even employing a heuristic algorithm. By breaking into a number of small ATAs, the entire FSM takes much less time to store into these ATAs. For example, the construction time of heuristic algorithm for the 3M name table can be reduced from days (for basic ATA) to minutes (for MATA), with an appropriate set of prime numbers.

2.5 Name table update

There are two types of name table updates: insertion and deletion. In this subsection, we demonstrate how these updates can be handled with ease in our design. First of all, it is worth reminding the reader that our name lookup mechanism is composed of two components.

(1) The name table is first organized into a character-trie, which is essentially a finite state machine.

(2) The character-trie is then transformed into MATA.

Accordingly, insertion and deletion of names are first conducted on the character-trie, in order to determine the modifications that need to be made to the character-trie structure. Then, we carry out the modifications in MATA. Following this logic, we explain name insertion and deletion as follows.

2.5.1 Name deletion

To delete a name P from the name table, we simply conduct a lookup of name P in the name table. If it is not matched, we determine that the proposed deletion operation is invalid, as name P does not actually exist in the name table.

Once name P is properly matched and comes to the leaf node representing itself, we then simply backtrack

towards the root. (This can be done by remembering all the nodes we have traversed along the path from the root to the leaf node.) Every node with one single valid transition will be deleted from the trie (each trie node contains a field that records the number of children nodes), till when we encounter the first node with next-hop information or more than one valid transition.

It is equally simple to carry out character-trie node deletion in MATA. Since every node to be deleted has one single valid transition, deleting the node is equivalent to deleting its stored valid transition in MATA. It takes one single memory access to locate the transition array element storing that transition, and mark that transition array element as vacant.

2.5.2 Name insertion

To insert a name P into the name table, we also conduct a lookup of name P in the name table, where we traverse the character-trie in a top-down manner, starting from the root. At each node on our way down the character-trie, if an existing transition is properly matched by P, we need to do nothing about the node. Otherwise, suppose we read in a number of characters from name P, which is converted into an integer x. We add a new transition on x to the current node, pointing to a new node we create for this new transition. This process continues until lookup of name P is completed.

To add an existing node's new transition on x into MATA, we directly locate the transition array element in which the new transition should be stored. If that element is vacant, we simply store the new transition into that element, and we are done. Otherwise, suppose the existing node needs to be relocated to resolve storage collision. This is done as if the node is a new node to be stored into MATA, following the algorithm described in Section 2.2 and 2.4. (One minor difference is that, here we also need to update the upstream transition pointing to the relocating node. This can be easily handled by always remembering the parent node of current node, during the lookup process for name P.)

3 The CPU-GPU System: Packet Latency and Stream Pipeline

GPU achieves high processing throughput by exploiting massive data-level parallelism — a large batch of names are processed by a large number of GPU threads concurrently. However, this massive batch processing mode can lead to extended per packet lookup latency[5]. Figure 4 presents a 4-stride MATA's throughput and latency

[5]Name lookup engine latency: the aggregate time from a packet being copied from host CPU to GPU device till its lookup result being returned from GPU to CPU.

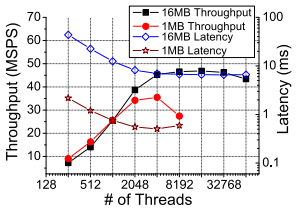


Figure 4: Throughput and latency of MATA without pipeline (3M name table, average workload).

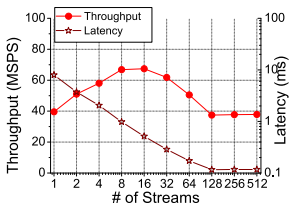


Figure 5: Throughput and latency of MATA with pipeline (3M name table, average workload).

obtained on the 3M name table, where names are processed in 16MB batches. As we can see, per-packet lookup latency can be many milliseconds. While in practice, telecommunication industry standards require that the entire system latency should be less than 450 μs[6]; name lookup as one of the various packet processing tasks should take no longer than 100 μs.

This extended lookup latency results from concurrent lookups of multiple names, due to contention among concurrent lookup threads processing different names. That said, a straightforward thought, therefore, is to reduce per-packet lookup latency by reducing batch size. Figure 4 also presents the above MATA's throughput and latency obtained with 1MB batch size. As we can see, small batch size leads to reduced lookup throughput, due to reduced data-level parallelism. But it is insufficient to hide off-chip DRAM access latency, causing throughput decline accordingly. Essentially, this latency-throughput dilemma is rooted in the GPU design philosophy of exploiting massive data-level parallelism. Unfortunately, previous proposals on GPU-based pattern matching (e.g. [29, 16]) have not taken latency requirements into account.

In this work, we resolve this latency-throughput dilemma by exploiting the multi-stream mechanism featured in NVIDIA's Fermi GPU architecture. In CUDA programming model, a *stream* is a sequence of operations that execute in issue-order. For example, in our design, each stream is composed of a number of lookup threads, each thread consisting of three tasks. (1) `DataFetch`: copy input names from host CPU to GPU device (via PCIe bus); (2) `Kernel`: perform name lookup inside GPU; (3) `WriteBack`: write lookup results back from GPU device to host CPU (via PCIe bus). Among them, `DataFetch` and `WriteBack` tasks are placed into one queue, executed by the *copy engine*; `Kernel` tasks are organized into another queue, executed by the *kernel engine*. Tasks in the same queue are executed in the order they enter the queue. In our design, each batch of input names is divided into m subsets; the kth subset is assigned to the kth stream for lookup.

[6]Here we refer to the specifications of the ISDN switch.

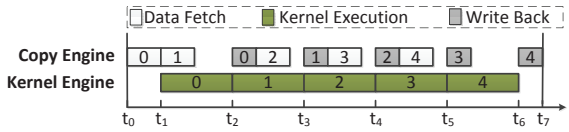


Figure 6: Multi-stream pipeline solution.

By pipelining these concurrent streams, lookup latency can be effectively reduced while keeping high lookup throughput. Algorithm 1 shows the pseudo code description of this optimized scheduling.

As shown in Figure 6, the `Kernel` task of stream *i* runs (on the kernel engine) in parallel with the `WriteBack` task of stream *i*-1 followed by the `DataFetch` task of stream *i*+1 (both running on the copy engine).

Figure 5 presents MATA's throughput and latency obtained on the 3M name table with 16MB batch size organized into 1∼512 streams, using 2,048 threads. This multi-stream pipeline solution successfully reduces lookup latency to 101μs while maintaining lookup throughput (using 128 or more streams).

Throughput: As described in Section 6.2.2, the copy engine is the throughput bottleneck. Then, the time T to finish processing an input name batch can be calculated by Formula (1) [15].

$$T = 2t_{start} * N + \frac{M_{batch} + M_{result}}{S_{PCIe}} + \frac{M_{batch}}{N * S_{kernel}} \quad (1)$$

Here, M_{batch} and M_{result} are the name batch size and the corresponding lookup results size, respectively. S_{PCIe} is the PCIe speed, S_{kernel} is the name lookup speed in GPU kernel, t_{start} is the warm up time of the copy engine, and N is the number of streams. Maximizing throughput means minimizing T. According to Fermat's theorem [2], T gets the minimal value at the stationary point $f'(N) = 0$, where $N = \sqrt{\frac{M_{batch}}{2t_{start} * S_{kernel}}}$. In Figure 5, our CPU-GPU name lookup engine has M_{batch}=16MB, t_{start}≈10μs, S_{kernel}≈8GB/s (200MSPS × 40B/packet). So the engine gets the maximal throughput with N=16 streams.

Algorithm 1 Multi-stream Pipeline Scheduling

```
1: procedure MultiStreamPipelineScheduling
2:    i ← 0;
3:    offset ← i*data_size/m;
4:    DataFetch(offset, streams[i]);
5:    Kernel(offset, streams[i]);
6:    for i:  0 → m-2 do
7:       offset ← (i+1)*data_size/m;
8:       DataFetch(offset, streams[i+1]);
9:       Kernel(offset, streams[i+1]);
10:       wb_offset ← i*data_size/m;
11:       WriteBack(wb_offset, streams[i]);
12:    end for
13:    WriteBack(offset, streams[m-1]);
14: end procedure
```

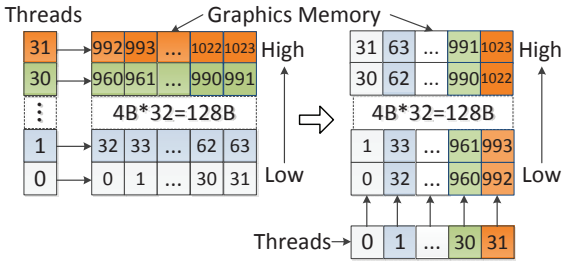


Figure 7: Input name storage layout.

Latency: Lookup latency $T_{latency}$ equals to the period from a stream's `DataFetch` task launched to the corresponding `WriteBack` task finished, i.e.,

$$T_{latency} = 2t_{start} + \frac{1}{N} * \left(\frac{M_{batch} + M_{result}}{S_{PCIe}} + \frac{M_{batch}}{S_{kernel}} \right) \quad (2)$$

Obviously, lookup latency decreases as the increasing of the stream number N.

4 Memory Access Performance

Like in all other modern computing architectures, memory access efficiency has a significant impact on GPU application performance. One practical approach to boosting performance is to reduce the amount of slow DRAM accesses, by exploiting GPU's memory access coalescence mechanism. In NVIDIA GeForce GTX GPU devices, the off-chip DRAM (e.g. global memory) is partitioned into 128-byte memory blocks. When a piece of data is requested, the entire 128-byte block containing the data is fetched (with one memory access). When multiple threads simultaneously read data from the same block, their read requests will be coalesced into one single memory access (to that block).

In our design, we employ an effective technique for optimizing memory access performance called input *interweaving* [32], which stores input names in an interweaved layout. In NVIDIA GeForce GTX GPUs, every 32 threads (with consecutive thread IDs) are bundled together as a separate *warp*, running synchronously in a single-instruction multiple-data (SIMD) manner — at any time, the 32 threads execute the same instruction, on possibly different data objects. In common practice, input data (i.e., names) are simply stored contiguously, as shown in the left part of Figure 7. For ease of illustration, each name is 128-byte long and occupies a whole memory block (one row). The 32 names parallel processed by the 32 threads in a *warp* reside in 32 different 128-byte blocks. Therefore, when the 32 threads simultaneously read the first piece of data from each of the names they are processing, resulting in 32 separate memory accesses that cannot be coalesced.

Here, memory access performance can be substantially improved by storing input names in an interweaved layout. Suppose the name lookup engine employs N_{thread} concurrent GPU threads. (1) Host CPU distributes input names into N_{thread} queues (i.e., name sequences), in the order of their arrival[7]; (2) Then, every 32 adjacent name sequences are grouped together, to be processed by a GPU warp consisting of 32 threads; each thread in the warp locates one of the 32 name sequence using its thread ID; (3) Finally, each group of 32 name sequences are interweaved together. Iteratively, CPU takes a 4-byte data *slice* from the head of each name sequence and interweaves them into a 128-byte memory block (one row), as shown in the right part of Figure 7. After interweaving, the 32 data slices of one name are stored in 32 different blocks (one column), and the 32 data slices belonging to 32 different names are stored in one block. The interweaved memory blocks are then transmitted to GPU's DRAM. Now, when the 32 threads in the same warp each requests for a slice from its own name sequence simultaneously, the 32 requested slices reside in the same 128-byte block. Therefore, the 32 memory requests are coalesced into one single memory access to that block. Interweaving thus significantly reduces the total amount of memory accesses to DRAM and hence substantially boosts overall performance.

5 Implementation

In this section, we describe the implementation of our GPU-based name lookup engine, including its input/output. First in Section 5.1, we introduce the hardware platform, operating system environment and development tools with which we implement the name lookup engine. Then in Section 5.2, we present the framework of our system. The lookup engine has two inputs: name tables and name traces. We introduce how we obtain or generate the name tables and name traces in Section 5.3 and Section 5.4, respectively.

5.1 Platform, environment and tools

We implement and run the name lookup engine on a commodity PC installed with an NVIDIA GeForce GTX 590 GPU board. The PC is installed with two 6-core CPUs (Intel Xeon E5645×2), with 2.4GHz clock frequency. Relevant hardware configuration is listed in Table 1.

Table 1: Hardware configuration.

Item	Specification
Motherboard	ASUS Z8PE-D12X (INTEL S5520)
CPU	Intel Xeon E5645×2 (6 cores, 2.4GHz)
RAM	DDR3 ECC 48GB (1333MHz)
GPU	NVIDIA GTX590 (2×512 cores, 2×1536MB)

The PC runs Linux Operating System version 2.6.41.9-1.fc15.x86_64 on its CPU. The GPU runs CUDA NVIDIA-Linux operating system version x86_64-285.05.09.

The entire lookup engine program consists of about 8,000 lines of code, and is composed of two parts: a CPU-based part and a GPU-based part. The CPU part of the system is developed using the C++ programming language; the GPU part of the system is developed using NVIDIA CUDA C programming language's SDK 4.0.

5.2 System framework

Figure 8 depicts the framework of our name lookup engine. Module 1 takes a name table as input and builds a character-trie for aggregating and representing that name table; this character-trie serves as the control plane of name lookup. To implement high speed and memory efficient name lookup, Module 2 transforms the character-trie into MATA, which serves as the data plane of name lookup, and hence will be transferred to and operated in the GPU-based lookup engine. Module 3 is the lookup engine operating in GPU, which accepts input names, performs lookup in the MATA and outputs lookup results. Meanwhile, GPU takes the generated name traces as input to search against the name table, which is implemented as the MATA. The lookup result in GPU is output to a module running on CPU, which obtains next-hop interface information based on GPU's output. There is also a 5-th module that is responsible for handling name table updates. We measure the core latency between point *A* and *B*, that is from the sending buffer to the receiving buffer of CPU.

5.3 Name Tables

The two name tables used in our experiments contain 2,763,780 entries and 10,000,000 entries, respectively. For brevity, we refer to them as the "3M name table" and the "10M name table", respectively. Each name table entry is composed of an NDN-style name and a next hop port number. As NDN is not yet standardized and there is no real NDN network, the name tables are obtained through the following five-step process.

[7]When appending names into sequences, we transform each input name (and name sequence) from a sequence of characters into a sequence of numbers. In our implementation and experiments, we implement 4-stride FSMs and hence each input name is transformed into a sequence of 32-bit `unsigned int` type integers. To transform an input name, CPU keeps reading up to 4 characters from that name unless a '/' is encountered; the characters read out are then transformed into an `unsigned int` integer. Each name sequence is thus transformed into a sequence of 32-bit integers.

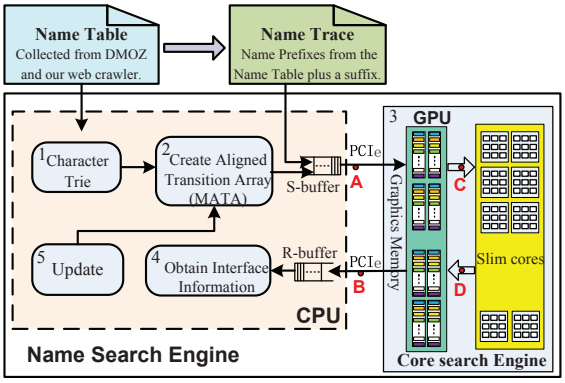


Figure 8: Framework of the name lookup engine.

Step 1: We collect Internet domain names in two ways. (1) We obtain existing domain name information from DMOZ [1], which is later used to generate the 3M name table. (2) We use a web crawler program to collect domain names, which are later used to generate the 10M name table. To achieve good geographic coverage, we ran web crawler programs on three servers located in North America, Europe and Asia, respectively. The web crawler programs kept collecting URLs from October 1st, 2011 to March 31st, 2012. At last, the crawler collected 7M domain names different from that collected from DMOZ. Consequently, we obtain 10 million non-duplicate domain names in total with our maximum efforts.

Step 2: We convert the domain names into NDN-style names, by putting the components in reverse order. For example, domain name `www.parc.com` is transformed into `/com/parc/www/`.

Step 3: For each NDN-style name, we map its corresponding domain name to an IP address resolved by DNS.

Step 4: For each NDN-style name, we obtain its next hop port number by performing longest prefix matching on its IP address obtained in Step 3, using an IP FIB downloaded from www.ripe.net.

Step 5: We map each NDN prefix to its obtained next hop port number, which gives us the final name table.

5.4 Name Traces

The name traces, which are generated from name tables, simulate the destination names carried in NDN packets. The names are formed by concatenating name prefixes selected from the name table and randomly generated suffixes. We generate two types of name traces, simulating average lookup workload and heavy lookup workload, respectively. Each name trace contains 200M names, and is generated as follows.

For each name table used in our experiments, its average workload trace is generated by randomly choosing names from the name table; its heavy work load trace is generated by randomly choosing from the top 10% longest names in the name table. Intuitively, the longer the input names are, the more state transition operations the GPU will perform for their lookup, meaning heavier workload.

Names chosen from name tables do not have a directory path. From the URLs we collected from Internet, we randomly choose a directory path for each name in the traces and append that path to the name.

6 Experimental Evaluation

We compare the performance of four lookup methods we have discussed in Section 2. The baseline method is using a two-dimensional state transition table (denoted by STT), which is largely compressed by ATA. Then, we upgrade ATA into 4-stride MATA. Finally, we improve MATA with interweaved name storage (denoted by MATA-NW).

First, in Section 6.1, we evaluate the memory efficiency of these methods. Then in Section 6.2, we conduct comprehensive evaluation and analysis of the throughput and latency of our name lookup engine — both the entire prototype engine and the core GPU part. The scalability of our lookup engine is evaluated in Section 6.3. Finally in Section 6.4, we evaluate its performance on handling name table updates.

6.1 Memory Space

If implemented with STT, the 3M and 10M name tables will take 19.49GB and 69.62GB, respectively. Compared with state transition table, ATA compresses storage space by 101× (on the 3M name table) and 102× (on the 10M name table), respectively. By constructing multiple smaller ATAs, MATA further compresses storage space — **MATA compresses storage space by 130× (on the 3M name table) and 142× (on the 10M name table), respectively**, easily fitting into the GTX590 GPU board of our prototype system, which has 3GB off-chip DRAM on board. (Note that input name interweaving changes storage layout but does not change storage space requirement.)

6.2 Lookup Performance

6.2.1 CPU-GPU System Performance

We now proceed to compare the lookup throughput and latency that can be achieved by the methods in comparison. Due to the excessive memory space requirement of STT, we hereby implement two small subsets of the 3M and 10M name tables, respectively, each containing 100,000 name entries. To explore the best achievable

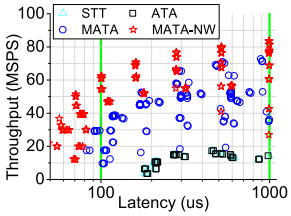


Figure 9: Throughput and latency on the 3M table's subset (average workload).

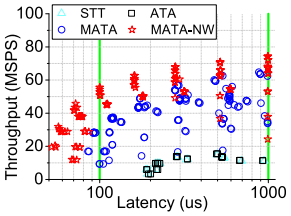


Figure 10: Throughput and latency on the 3M table's subset (heavy workload).

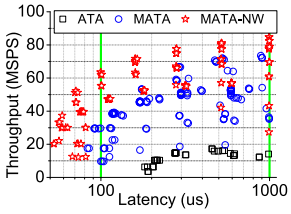


Figure 13: Throughput and latency on the 3M table (average workload).

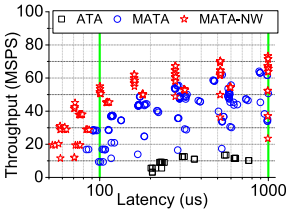


Figure 14: Throughput and latency on the 3M table (heavy workload).

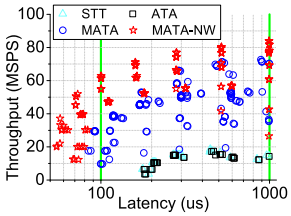


Figure 11: Throughput and latency on the 10M table's subset (average workload).

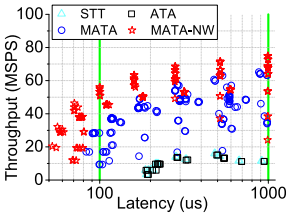


Figure 12: Throughput and latency on the 10M table's subset (heavy workload).

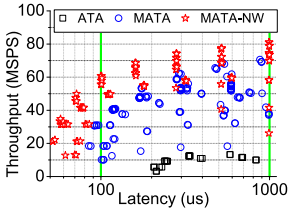


Figure 15: Throughput and latency on the 10M table (average workload).

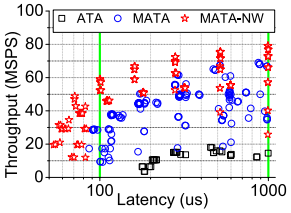


Figure 16: Throughput and latency on the 10M table (heavy workload).

lookup performance of each method, we run experiments with a wide range of parameter settings: doubling number of CUDA thread blocks from 8 to 4096, doubling number of threads per CUDA thread block from 32 to 1024, and doubling CUDA stream count from 1 to 4096. The measured lookup throughput and latency of the four methods are plotted in Figure 9-12, in which one point means the throughput and latency of the method with one parameter setting. (For legibility, we have only plotted results with less than 1ms latency.)

As we expected, STT and ATA have nearly identical performance, although ATA uses two orders of magnitude less memory. With multi-striding, MATA significantly outperforms ATA. STT and ATA have not been able to meet the 100μs latency requirement; in contrast, MATA can achieve up to 29.75 MSPS under average worload and 28.52 MSPS under heavy workload, while keeping latency below 100μs. With input name interweaving, MATA-NW further raises lookup throughput to 61.40 MSPS under average workload and 56.10 MSPS under heavy workload. The minimum lookup latency that can be achieved by MATA-NW is around 50μs (with about 20 MSPS lookup throughput), while the minimum achievable latency of STT and ATT is around 200μs. With a 200μs latency requirement, the maximum throughput that can be achieved by STT, ATA, MATA and MATA-NW are 6.59, 6.56, 52.99 and 71.12 MSPS under average workload and 6.02, 6.01, 49.04 and 63.32 MSPS under heavy workload, respectively; **MATA-NW achieves over 10× speedup.**

As the above two subsets are relatively small, we then conduct experiments based on the 3M and 10M name tables (without STT), and plot the results in Figure 13-16. The results are similar to what we observe on the two subsets. With 100μs latency requirement, MATA-NW can achieve 63.52 MSPS under average workload and 55.65 MSPS under heavy workload, translating to 127 Gbps under average workload and 111 Gbps under heavy workload, respectively.

6.2.2 GPU Engine Core Performance

The above experiments have not answered the following question — *which part of the prototype system is the performance bottleneck?* To answer this question, we conduct the following experiments for comparison[8]. (1) In experiment I, the GPU does not perform lookup on any input name and directly returns. The measured throughput represents the raw bandwidth of PCIe bus connecting CPU and GPU, and is plotted as "PCIe" in Figure 17-18; (2) In experiment II, the GPU performs one lookup on every input name. The measured throughput represents the normal throughput of our prototype system and is plotted as "MATA-NW". The experiment results reveal that system throughput is tightly bounded by PCIe bus bandwidth; although with overly large stream counts, system throughput starts dropping due to insufficient number of threads per stream[9].

As the PCIe bus limits the lookup throughput achievable with our prototype system, we conduct another set of experiments to evaluate the lookup throughput that can be achieved with our core algorithmic design (MATA-

[8]Here, lookup throughput is obtained with 16 concurrent CUDA thread blocks and 32 threads per CUDA thread block running in parallel. Because the above experiment results show that they produce the best lookup throughput under 100μs latency requirement.

[9]Recall that every warp of 32 threads must execute synchronously.

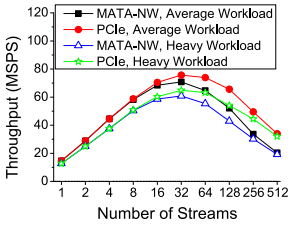


Figure 17: System throughput on the 3M table.

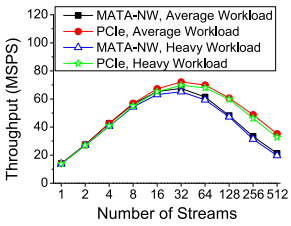


Figure 18: System throughput on the 10M table.

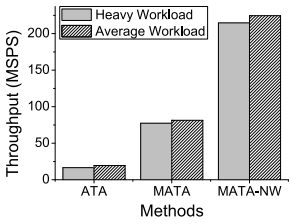


Figure 19: Kernel throughput on the 3M table.

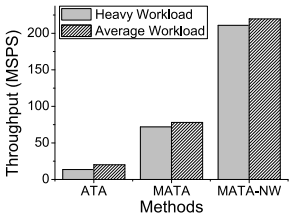


Figure 20: Kernel throughput on the 10M table.

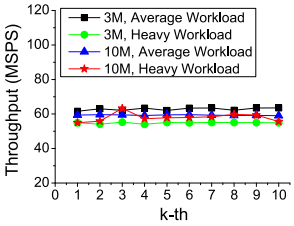


Figure 21: Growth trend of lookup throughput.

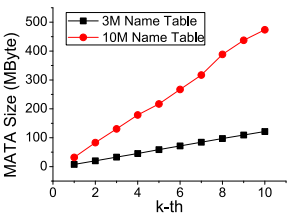


Figure 22: Growth trend of MATA size.

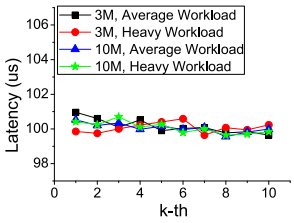


Figure 23: Growth trend of lookup latency.

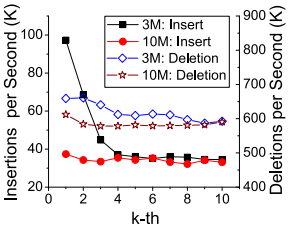


Figure 24: Growth trend of update performance.

NW) running on GPU. (1) First, we transmit input names from CPU to GPU's global memory in advance; (2) Then, we perform name lookup on GPU. Lookup results are not written back to CPU (via PCIe). When calculating lookup throughput, we do not include the time taken in (1). The calculated throughput represents the GPU's kernel throughput, without the bottleneck of PCIe bandwidth. As we can see in Figure 19-20, GPU's kernel throughput is more than two times higher than the entire engine throughput, reaching up to 219.69 MSPS, which is 96× of CPU-based implementation (MATA can perform 2.28 MSPS with one thread in CPU-based platform). These results demonstrate the real potential of our GPU-based name lookup engine design; this potential can be realized by high speed routers, which do not have the PCIe bandwidth problem.

6.3 Scalability

While our GPU-based name lookup engine is demonstrated to perform well on the 3M and 10M name tables, we are also interested in foreseeing its performance trend as name table size grows. For that, we partition each name table into ten equal-sized subsets, and progressively generate ten name tables for each of them; the kth generated name table consists of the first k equal-size subsets. Experiments are then conducted on these 20 generated name tables derived from the 3M name table and 10M name table. Measured results on lookup throughput, memory space requirement and lookup latency are presented in Figure 21-23, respectively.

As name table size grows, lookup throughput and lookup latency tend to stabilize around 60 MSPS and 100μs, respectively. The memory space requirement, represented by MATA size, tends to grow with linear scalability, which is consistent with our intuition.

6.4 Name table update

Finally, we measure the performance of our design on handling name table updates, both insertions and deletions. The results are reported in Figure 24. The general trend is that, the larger the name table, the more difficult it is to handle updates. Just like what we have observed on the growth trend of throughput and latency, update performance also tends to stabilize at a certain performance level. On both name tables, we can consistently handle more than 30K insertions per second. As we have described in Section 2, deletions are much easier to implement than insertions; we can steadily handle around 600K deletions per second. Compared with the current IP networks, which have an average update rate of several thousand per second, our name updating mechanism runs one order of magnitude faster.

7 Related Work

7.1 GPU-based Packet Processing

GPU as a high throughput parallel processing platform is attracting proliferating interest, and is being studied as a potential platform for high speed packet processing, such as IP lookup [10, 30, 19], packet classification [14, 20] and pattern matching [17, 7, 29, 24, 32, 16].

Pattern matching is a much simpler special form of name lookup, whose technical core is longest prefix string matching. While the initial study on name-based routing [22, 11, 25, 26] has revealed the feasi-

bility of routing based on hierarchical names instead of IP addresses, there has been lacking a comprehensive implementation-based study to address the following practical problems: (1) With large-scale name tables containing millions of names, how and to what extent can name tables be compressed for practical implementation; (2) What name lookup throughput can be reached under practical latency constraints; (3) What update performance can be obtained.

The existing methods for GPU-based pattern matching [17, 7, 29, 24, 32, 16] are designed targeting IDS-like systems where packet latency is not a first priority, and have all ignored the important issue of packet latency. As we have analyzed in Section 3 and demonstrated through experiments in Section 6, the latency-throughput tradeoff is rooted in GPU's design philosophy; optimizing throughput without considering latency constraints leads to overly optimistic results, and are not practically competent for high speed routers. By employing the multi-stream mechanism featured by NVIDIA Fermi architecture, which has been proven effective in other fields (e.g. [21, 13, 6]), our design is able to achieve wire speed lookup throughput with 50-100μs packet latency.

In fact, the existing GPU-based pattern matching methods have not even considered practical performance issues specific to such CPU-GPU hybrid systems, such as data transmission (e.g. via PCIe bus). Meanwhile, the existing methods have also not paid deserved attention to update performance, which is an important issue in high speed router design. The pattern sets used in their study are also multiple orders of magnitude smaller than what we target and have adopted in our experiments. On one hand, this requires more advanced compression techniques; on the other hand, high speed lookup can become even more challenging in the presence of more sophisticated compression. In contrast, our work is the first system-based study on GPU-based large-scale pattern matching and addresses all these performance issues: lookup throughput, latency, memory efficiency, update performance and CPU-GPU communication.

IP lookup is much simpler than name lookup. For example, in our implementation-based study, average name length is around 40 bytes, 10$\times$ longer than IP addresses used in GPU-based IP lookup research (e.g. PacketShader [10]). Unlike fixed length IP addresses, names are also variable in length, making it even more complex to implement efficient lookup. Moreover, name tables are 1-2 orders of magnitude larger than IP forwarding tables in terms of entry count, and 2-3 orders of magnitude larger in terms of byte count.

Packet classification is more complex than IP lookup in that packet classification rules typically check five packet header fields (13 bytes in total for IPv4) including two IP addresses. Nevertheless, packet classification rules are still much shorter than names, and are also fixed in length. In fact, packet classification rule sets are typically 1-3 orders of magnitude smaller than IP forwarding tables, let alone name tables.

In summary, GPU-based IP lookup and packet classification are not comparable/applicable to the large-scale name lookup problem studied in our work. Meanwhile, almost all of these GPU-based packet processing techniques (except PacketShader [10]) ignore the important issue of packet processing latency.

7.2 Algorithm & Data Structure

The technical core of name lookup is longest prefix matching. Before determining the matched longest prefix, hash-based methods have to perform multiple hash computations, which significantly degrade lookup performance [27]. In trie-based methods, to quickly determine the correct branch to transfer, hash techniques have been intensively studied. B. Michel et al. designed an incremental hash function called Hash Chain [18], improved by Zhou et al. [31], to aggregate URLs sharing common prefixes, while minimizing collisions between prefixes. These hash-based algorithms all have a common drawback — false positives due to hash collisions. More importantly, hash collision during trie traversal can lead name lookup to a wrong branch, causing packets to be forwarded to wrong ports; this undermines routing integrity — a fundamental property of NDN routers. So remedies for false positives are required in these systems. For eliminating false positive, we [26] proposed to encode all the name components for efficient traversal. When balancing hash collision probability and memory space requirement, we may not necessarily gain memory efficiency.

Tarjan and Yao proposed in their pioneer work [23] a method for compressing two-dimensional (state transition) tables into compact one-dimensional arrays — the original idea underlying basic ATA. In a follow-up work, Suwaiyel and Horowitz [5] proved that producing an optimal array is NP-hard, and proposed approximation algorithms for producing arrays. Our work develops beyond prior arts on the following aspects: (1) We propose a multi-stride ATA approach that can significantly improve storage efficiency, matching speed and lookup latency; (2) We propose the more advanced multi-ATA (MATA) design. Compared with the basic ATA design, MATA liberates multi-stride ATA from the constraint of available memory space. As a result, matching speed, latency, storage space and array construction time are all optimized substantially. We demonstrate through experimental evaluation that, while the naïve ATA design can be inadequate for large-scale, high throughput and low latency name lookup, the innovated MATA design is suf-

ficient for tackling such real applications.

7.3 Software Routers

Compared with hardware routers, software routers are cost-effective and much easier to program. Software routers can also provide extensible platforms for implementing CCN router prototypes.

PacketShader [10] exploits the GPU's massively-parallel processing power to overcome the CPU bottleneck in the current software IP routers. However, name lookup is more complex than IP lookup, for the variable and unbounded length of names as well as the much larger name tables. Compared with the IP FIB in [10], name FIB in our engine needs elaborate data structures to reduce memory consumption and speed up name lookup. Besides, PacketShader has not described any detail about how to balance lookup throughput and latency in PacketShader. However, the high performance packet I/O engine in PacketShader may help improve the performance of a whole CCN router.

In backbone routers, a high speed interface (e.g. 40 Gbps OC-768) is usually processed by a single data plane. RouteBricks [8] bundles multiple PCs together to handle packets, but one single PC is hard to handle the traffic from a high speed interface. Our work focuses on wire speed name lookup with a single data plane, which is different from RouteBricks. If more capacity or a larger number of ports is needed, we can apply a multi-machine approach in [8].

8 Discussion and Conclusion

8.1 Discussion

As described in Section 6.2.2, PCIe is the bottleneck of our name lookup engine, which has to use a GPU board installed on a PC via PCIe. However, it does not rule out the possibility of embedding GPUs into a high-end router through other non-PCIe means. Moreover, note that, although GTX590 GPU has two processor cores on chip, we have only used one of them in our experiments; using both processor cores can potentially boost performance as well.

Since CCN has not been standardized yet, its FIB table size and name length bound are still unknown. The 10M name table, collected with our maximum efforts, has only consumed one-sixth of the memory resource of GTX590. Thus we estimate our name lookup engine at least can handle a name table with 60M prefixes while keeping the lookup throughput. However, the name table would be orders of magnitude vaster when the prefixes cannot be aggregated effectively. As the first step, in this paper we demonstrate the feasibility of implementing wire speed name lookup on a table of substantial size. Scaling name tables to the larger conceivable size will be our future work.

Routing changes, including network topology changes, routing policy modifications and content publish/deletion, will cause FIB updates. Given no CCN/NDN network is deployed today, FIB update frequency cannot be accurately measured. We will estimate the update frequency from the current Internet. On one hand, the frequency of network topology changes and routing policy modifications in CCN can be inferred from the current IP network, which makes up to several thousand updates per second. On the other hand, content publish/deletion will not necessarily lead to FIB updates, since the name prefixes in FIBs are aggregated. If we assume the addition and deletion of domains will cause FIB updates, there are only a few tens of FIB updates per second, according to the top-level domain statistics [4] of the current Internet. Therefore, in this case, our approach can meet update performance requirements. Whether our FIB update mechanism meets practical performance requirements in real CCN networks needs to be further studied in future work.

8.2 Conclusion

Name lookup is one of the fundamental functions underlying a great variety of technological fields, among which wire speed CCN name lookup in large-scale name tables is the most challenging. Thus far, implementation-based study on real systems has been lacking. How much memory will CCN name tables consume? How fast can name lookup engine be implemented, from both technical and economic point of view? These are still unknown. In this paper, we propose a multi-stride character-trie algorithm, implemented in our GPU-based name lookup engine with a number of new techniques. Extensive experiments demonstrate that our GPU-based lookup engine can achieve up to 63.52 MSPS on name tables containing millions of names under a delay of less than 100μs. Our GPU-based implementation results answer that large-scale name lookup can not only run at high speed with low latency and fast incremental update, but also be cost-effective with today's off-the-shelf technologies.

9 Acknowledgments

We thank the anonymous reviewers and our shepherd Michael Walfish for their help and invaluable comments.

References

[1] DMOZ - Open Directory Project. http://www.dmoz.org/.

[2] Fermat's theorem (stationary points). http://en.wikipedia.org/wiki/fermat's_theorem_(stationary_points).

[3] http://s-router.cs.tsinghua.edu.cn/namelookup.org/index.htm.

[4] Internet Statistics, http://www.whois.sc/internet-statistics/.

[5] AL-SUWAIYEL, M., AND HOROWITZ, E. Algorithms for trie compaction. *ACM Trans. Database Syst. 9*, 2 (June 1984), 243–263.

[6] BHATOTIA, P., RODRIGUES, R., AND VERMA, A. Shredder: GPUaccelerated incremental storage and computation. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)* (2012).

[7] CASCARANO, N., ROLANDO, P., RISSO, F., AND SISTO, R. iNFAnt: NFA pattern matching on gpgpu devices. *SIGCOMM Comput. Commun. Rev. 40*, 5, 20–26.

[8] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B.-G., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. RouteBricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (New York, NY, USA, 2009), SOSP'09, ACM, pp. 15–28.

[9] FREDKIN, E. Trie memory. *Commun. ACM 3*, 9 (Sept. 1960), 490–499.

[10] HAN, S., JANG, K., PARK, K., AND MOON, S. PacketShader: a GPU-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM* (New York, NY, USA, 2010), SIGCOMM '10, ACM, pp. 195–206.

[11] HWANG, H., ATA, S., AND MURATA, M. A Feasibility Evaluation on Name-Based Routing. In *IP Operations and Management*, G. Nunzi, C. Scoglio, and X. Li, Eds., vol. 5843 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2009, pp. 130–142.

[12] JACOBSON, V., SMETTERS, D. K., THORNTON, J. D., PLASS, M. F., BRIGGS, N. H., AND BRAYNARD, R. L. Networking Named Content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies* (2009), CoNEXT '09, ACM, pp. 1–12.

[13] JANG, K., HAN, S., HAN, S., MOON, S., AND PARK, K. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2011).

[14] KANG, K., AND DENG, Y. Scalable packet classification via GPU metaprogramming. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011* (march 2011), pp. 1 –4.

[15] L. HENNESSY, J., AND A. PATTERSON, D. *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann Publishers, 2011.

[16] LIN, C.-H., LIU, C.-H., CHANG, S.-C., AND HON, W.-K. Memory-efficient pattern matching architectures using perfect hashing on graphic processing units. In *INFOCOM, 2012 Proceedings IEEE* (march 2012), pp. 1978 –1986.

[17] LIN, C.-H., TSAI, S.-Y., LIU, C.-H., CHANG, S.-C., AND SHYU, J.-M. Accelerating String Matching Using Multi-Threaded Algorithm on GPU. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE* (dec. 2010), pp. 1 –5.

[18] MICHEL, B., NIKOLOUDAKIS, K., REIHER, P., AND ZHANG, L. URL forwarding and compression in adaptive Web caching. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* (2000), vol. 2, pp. 670 –678.

[19] MU, S., ZHANG, X., ZHANG, N., LU, J., DENG, Y. S., AND ZHANG, S. IP routing processing with graphic processors. In *Proceedings of the Conference on Design, Automation and Test in Europe* (2010), DATE '10, pp. 93–98.

[20] NOTTINGHAM, A., AND IRWIN, B. Parallel packet classification using GPU co-processors. In *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists* (New York, NY, USA, 2010), SAICSIT '10, ACM, pp. 231–241.

[21] RENNICH, S. C/C++ Streams and Concurrency. http://developer.download.nvidia.com/.

[22] SHUE, C., AND GUPTA, M. Packet Forwarding: Name-based Vs. Prefix-based. In *IEEE Global Internet Symposium, 2007* (May 2007), pp. 73 –78.

[23] TARJAN, R. E., AND YAO, A. C.-C. Storing a sparse table. *Commun. ACM 22*, 11 (Nov. 1979), 606–611.

[24] VASILIADIS, G., ANTONATOS, S., POLYCHRONAKIS, M., MARKATOS, E., AND IOANNIDIS, S. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. vol. 5230. 2008, pp. 116–134.

[25] WANG, Y., DAI, H., JIANG, J., HE, K., MENG, W., AND LIU, B. Parallel Name Lookup for Named Data Networking. In *IEEE Global Telecommunications Conference (GLOBECOM)* (dec. 2011), pp. 1 –5.

[26] WANG, Y., HE, K., DAI, H., MENG, W., JIANG, J., LIU, B., AND CHEN, Y. Scalable Name Lookup in NDN Using Effective Name Component Encoding. In *IEEE 32nd International Conference on Distributed Computing Systems (ICDCS)* (june 2012), pp. 688–697.

[27] WANG, Y., PAN, T., MI, Z., DAI, H., GUO, X., ZHANG, T., LIU, B., AND DONG, Q. NameFilter: Achieving fast name lookup with low memory cost via applying two-stage bloom filters. In *INFOCOM mini-conference 2013. Proceedings. IEEE* (2013).

[28] ZHANG, L., ESTRIN, D., JACOBSON, V., AND ZHANG, B. Named Data Networking (NDN) Project. http://www.named-data.net/.

[29] ZHAO, J., ZHANG, X., WANG, X., DENG, Y., AND FU, X. Exploiting graphics processors for high-performance IP lookup in software routers. In *INFOCOM, 2011 Proceedings IEEE* (april 2011), pp. 301 –305.

[30] ZHAO, J., ZHANG, X., WANG, X., AND XUE, X. Achieving O(1) IP lookup on GPU-based software routers. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM* (New York, NY, USA, 2010), SIGCOMM '10, ACM, pp. 429–430.

[31] ZHOU, Z., SONG, T., AND JIA, Y. A High-Performance URL Lookup Engine for URL Filtering Systems. In *Communications (ICC), 2010 IEEE International Conference on* (may 2010), pp. 1 –5.

[32] ZU, Y., YANG, M., XU, Z., WANG, L., TIAN, X., PENG, K., AND DONG, Q. GPU-based NFA implementation for high speed memory efficient regular expression matching. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2012).

SoNIC: Precise Realtime Software Access and Control of Wired Networks

Ki Suh Lee, Han Wang, Hakim Weatherspoon
Computer Science Department, Cornell University
kslee,hwang,hweather@cs.cornell.edu

Abstract

The physical and data link layers of the network stack contain valuable information. Unfortunately, a systems programmer would never know. These two layers are often inaccessible in software and much of their potential goes untapped. In this paper we introduce SoNIC, Software-defined Network Interface Card, which provides access to the physical and data link layers in software by implementing them in software. In other words, by implementing the creation of the physical layer bitstream in software and the transmission of this bitstream in hardware, SoNIC provides complete control over the entire network stack in realtime. SoNIC utilizes commodity off-the-shelf multi-core processors to implement parts of the physical layer in software, and employs an FPGA board to transmit optical signal over the wire. Our evaluations demonstrate that SoNIC can communicate with other network components while providing realtime access to the entire network stack in software. As an example of SoNIC's fine-granularity control, it can perform precise network measurements, accurately characterizing network components such as routers, switches, and network interface cards. Further, SoNIC enables timing channels with nanosecond modulations that are undetectable in software.

1 Introduction

The physical and data link layers of the network stack offer untapped potential to systems programmers and network researchers. For instance, access to these lower layers can be used to accurately estimate available bandwidth [23, 24, 32], increase TCP throughput [37], characterize network traffic [19, 22, 35], and create, detect and prevent covert timing channels [11, 25, 26]. In particular, idle characters that only reside in the physical layer can be used to accurately measure interpacket delays. According to the 10 Gigabit Ethernet (10 GbE) standard, the physical layer is *always* sending either data or idle characters, and the standard requires at least 12 idle characters (96 bits) between any two packets [7]. Using these physical layer (PHY[1]) idle characters for a measure of interpacket delay can increase the precision of estimating available bandwidth. Further, by controlling interpacket delays, TCP throughput can be increased by reducing bursty behavior [37]. Moreover, capturing these idle characters from the PHY enables highly accurate traffic analysis and replay capabilities. Finally, fine-grain control of the interpacket delay enables timing channels to be created that are potentially undetectable to higher layers of the network stack.

Unfortunately, the physical and data link layers are usually implemented in hardware and not easily accessible to systems programmers. Further, systems programmers often treat these lower layers as a black box. Not to mention that commodity network interface cards (NICs) do not provide nor allow an interface for users to access the PHY in any case. Consequently, operating systems cannot access the PHY either. Software access to the PHY is only enabled via special tools such as BiFocals [15] which uses physics equipment, including a laser and an oscilloscope.

As a new approach for accessing the PHY from software, we present SoNIC, Software-defined Network Interface Card. SoNIC provides users with unprecedented flexible realtime access to the PHY from software. In essence, all of the functionality in the PHY that manipulate bits are implemented in software. SoNIC consists of commodity off-the-shelf multi-core processors and a field-programmable gate array (FPGA) development board with peripheral component interconnect express (PCIe) Gen 2.0 bus. High-bandwidth PCIe interfaces and powerful FPGAs can support full bidirectional data transfer for two 10 GbE ports. Further, we created and implemented optimized techniques to achieve not only high-performance packet processing, but also high-performance 10 GbE bitstream control in software. Parallelism and optimizations allow SoNIC to process multiple 10 GbE bitstreams at line-speed.

With software access to the PHY, SoNIC provides the opportunity to improve upon and develop new network research applications which were not previously feasible. First, as a powerful network measurement tool, SoNIC can generate packets at full data rate with minimal interpacket delay. It also provides fine-grain control over the interpacket delay; it can inject packets with no variance in the interpacket delay. Second, SoNIC accurately captures incoming packets at any data rate including the maximum, while simultaneously timestamping each packet with sub-nanosecond granularity. In other

[1] We use PHY to denote the physical layer throughout the paper.

words, SoNIC can capture exactly what was sent. Further, this precise timestamping can improve the accuracy of research based on interpacket delay. For example, SoNIC can be used to profile network components. It can also create timing channels that are undetectable from software application.

The contributions of SoNIC are as follows:

- We present the design and implementation of SoNIC, a new approach for accessing the entire network stack in software in realtime.
- We designed SoNIC with commodity components such as multi-core processors and a PCIe pluggable board, and present a prototype of SoNIC.
- We demonstrate that SoNIC can enable flexible, precise, and realtime network research applications. SoNIC increases the flexibility of packet generation and the accuracy of packet capture.
- We also demonstrate that network research studies based on interpacket delay can be significantly improved with SoNIC.

2 Challenge: PHY Access in Software

Accessing the physical layer (PHY) in software provides the ability to study networks and the network stack at a heretofore inaccessible level: It can help improve the precision of network measurements and profiling/monitoring by orders of magnitude [15]. Further, it can help improve the reliability and security of networks via faithful capture and replay of network traffic. Moreover, it can enable the creation of timing channels that are undetectable from higher layers of the network stack. This section discusses the requirements and challenges of achieving realtime software access to the PHY, and motivates the design decisions we made in implementing SoNIC. We also discuss the Media Access Control (MAC) layer because of its close relationship to the PHY in generating valid Ethernet frames.

The fundamental challenge to perform the PHY functionality in software is maintaining synchronization with hardware while efficiently using system resources. Some important areas of consideration when addressing this challenge include *hardware support, realtime capability, scalability and efficiency, precision, and a usable interface*. Because so many factors go into achieving realtime software access to the PHY, we first discuss the 10 GbE standard before discussing detailed requirements.

2.1 Background

According to the IEEE 802.3 standard [7], the PHY of 10 GbE consists of three sublayers: the Physical Coding Sublayer (PCS), the Physical Medium Attachment (PMA) sublayer, and the Physical Medium Dependent (PMD) sublayer (See Figure 1). The PMD sublayer is responsible for transmitting the outgoing symbolstream over the physical medium and receiving the incoming

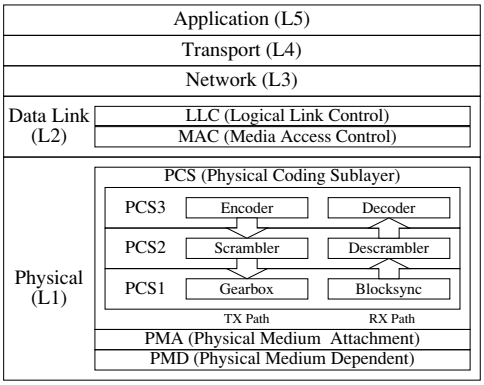


Figure 1: 10 Gigabit Ethernet Network stack.

symbolstream from the medium. The PMA sublayer is responsible for clock recovery and (de-)serializing the bitstream. The PCS performs the blocksync and gearbox (we call this PCS1), scramble/descramble (PCS2), and encode/decode (PCS3) operations on every Ethernet frame. The IEEE 802.3 Clause 49 explains the PCS sublayer in further detail, but we will summarize below.

When Ethernet frames are passed from the data link layer to the PHY, they are reformatted before being sent across the physical medium. On the transmit (TX) path, the PCS encodes every 64-bit of an Ethernet frame into a 66-bit *block* (PCS3), which consists of a two bit *synchronization header* (syncheader) and a 64-bit *payload*. As a result, a 10 GbE link actually operates at 10.3125 Gbaud ($10G \times \frac{66}{64}$). The PCS also scrambles each block (PCS2) to maintain DC balance[2] and adapts the 66-bit width of the block to the 16-bit width of the PMA interface (PCS1; the gearbox converts the bit width from 66- to 16-bit width.) before passing it down the network stack. The entire 66-bit block is transmitted as a continuous stream of *symbols* which a 10 GbE network transmits over a physical medium (PMA & PMD). On the receive (RX) path, the PCS performs block synchronization based on two-bit syncheaders (PCS1), descrambles each 66-bit block (PCS2) before decoding it (PCS3).

Above the PHY is the Media Access Control (MAC) sublayer of the data link layer. The 10 GbE MAC operates in full duplex mode; it does not handle collisions. Consequently, it only performs data encapsulation/decapsulation and media access management. Data encapsulation includes framing as well as error detection. A Cyclic Redundancy Check (CRC) is used to detect bit corruptions. Media access management inserts at least 96 bits (twelve `idle /I/` characters) between two Ethernet frames.

On the TX path, upon receiving a layer 3 packet, the MAC prepends a preamble, start frame delimiter (SFD), and an Ethernet header to the beginning of the frame. It also pads the Ethernet payload to satisfy a

[2]Direct current (DC) balance ensures a mix of 1's and 0's is sent.

minimum frame-size requirement (64 bytes), computes a CRC value, and places the value in the Frame Check Sequence (FCS) field. On the RX path, the CRC value is checked, and passes the Ethernet header and payload to higher layers while discarding the preamble and SFD.

2.2 Hardware support

The hardware must be able to transfer raw symbols from the wire to software at high speeds. This requirement can be broken down into four parts: a) Converting optical signals to digital signals (PMD), b) Clock recovery for bit detection (PMA), and c) Transferring large amounts of bits to software through a high-bandwidth interface. Additionally, d) the hardware should leave recovered bits (both control and data characters in the PHY) intact until they are transferred and consumed by the software. Commercial optical transceivers are available for a). However, hardware that simultaneously satisfies b), c) and d) is not common since it is difficult to handle 10.3125 Giga symbols in transit every second.

NetFPGA 10G [27] does not provide software access to the PHY. In particular, NetFPGA pushes not only layers 1-2 (the physical and data link layer) into hardware, but potentially layer 3 as well. Furthermore, it is not possible to easily undo this design since it uses an on-board chip to implement the PHY which prevents direct access to the PCS sublayer. As a result, we need a new hardware platform to support software access to the PHY.

2.3 Realtime Capability

Both hardware and software must be able to process 10.3125 Gigabits per second (Gbps) continuously. The IEEE 802.3 standard [7] requires the 10 GbE PHY to generate a continuous bitstream. However, synchronization between hardware and software, and between multiple pipelined cores is non-trivial. The overheads of interrupt handlers and OS schedulers can cause a discontinuous bitstream which can subsequently incur packet loss and broken links. Moreover, it is difficult to parallelize the PCS sublayer onto multiple cores. This is because the (de-)scrambler relies on state to recover bits. In particular, the (de-)scrambling of one bit relies upon the 59 bits preceding it. This fine-grained dependency makes it hard to parallelize the PCS sublayer. The key takeaway here is that everything must be efficiently pipelined and well-optimized in order to implement the PHY in software while minimizing synchronization overheads.

2.4 Scalability and Efficiency

The software must scale to process multiple 10 GbE bitstreams while efficiently utilizing resources. Intense computation is required to implement the PHY and MAC layers in software. (De-)Scrambling every bit and computing the CRC value of an Ethernet frame is especially intensive. A functional solution would require multiple duplex channels to each independently perform the CRC, encode/decode, and scramble/descramble computations at 10.3125 Gbps. The building blocks for the PCS and MAC layers will therefore consume many CPU cores. In order to achieve a scalable system that can handle multiple 10 GbE bitstreams, resources such as the PCIe, memory bus, Quick Path Interconnect (QPI), cache, CPU cores, and memory must be efficiently utilized.

2.5 Precision

The software must be able to precisely control and capture interpacket gaps. A 10 GbE network uses one bit per symbol. Since a 10 GbE link operates at 10.3125 Gbaud, each and every symbol length is 97 pico-seconds wide ($= 1/(10.3125 * 10^9)$). Knowing the number of bits can then translate into having a precise measure of time at the sub-nanosecond granularity. In particular, depending on the combination of data and control symbols in the PCS block[3], the number of bits between data frames is not necessarily a multiple of eight. Therefore, on the RX path, we can tell the exact distance between Ethernet frames in bits by counting *every bit*. On the TX path, we can control the data rate precisely by controlling the number of `idle` characters between frames: An idle character is 8 (or 7) bits and the 10 GbE standard requires at least 12 idle characters sent between Ethernet frames.

To achieve this precise level of control, the software must be able to access every bit in the raw bitstream (the symbolstream on the wire). This requirement is related to point d) from Section 2.2. The challenge is how to generate and deliver *every* bit from and to software.

2.6 User Interface

Users must be able to easily access and control the PHY. Many resources from software to hardware must be tightly coupled to allow realtime access to the PHY. Thus, an interface that allows fine-grained control over them is necessary. The interface must also implement an I/O channel through which users can retrieve data such as the count of bits for precise timing information.

3 SoNIC

The design goals of SoNIC are to provide 1) access to the PHY in software, 2) realtime capability, 3) scalability and efficiency, 4) precision, and 5) user interface. As a result, SoNIC must allow users realtime access to the PHY in software, provide an interface to applications, process incoming packets at line-speed, and be scalable. Our ultimate goal is to achieve the same flexibility and control of the entire network stack for a wired network, as a software-defined radio [33] did for a wireless network, while maintaining the same level of precision as BiFocals [15]. Access to the PHY can then enhance the accuracy of network research based on interpacket delay.

[3] Figure 49-7 [7]

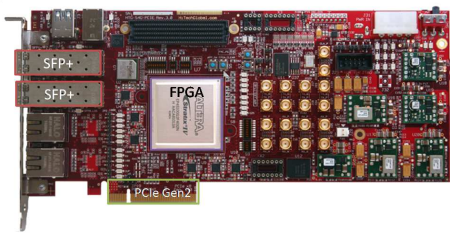
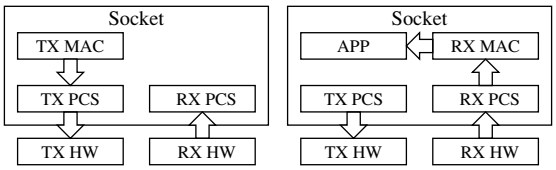


Figure 2: An FPGA development board [6]



(a) Packet Generator (b) Packet Capturer

Figure 3: Example usages of SoNIC

In this section, we discuss the design of SoNIC and how it addresses the challenges presented in Section 2.

3.1 Access to the PHY in software

An application must be able to access the PHY in software using SoNIC. Thus, our solution must implement the bit generation and manipulation functionality of the PHY in software. The transmission and reception of bits can be handled by hardware. We carefully examined the PHY to determine an optimal partitioning of functionality between hardware and software.

As discussed in Section 2.1, the PMD and PMA sublayers of the PHY do not modify any bits or change the clock rate. They simply forward the symbolstream/bitstream to other layers. Similarly, PCS1 only converts the bit width (gearbox), or identifies the beginning of a new 64/66 bit block (blocksync). Therefore, the PMD, PMA, and PCS1 are all implemented in hardware as a forwarding module between the physical medium and SoNIC's software component (See Figure 1). Conversely, PCS2 (scramble/descramble) and PCS3 (encode/decode) actually manipulate bits in the bitstream and so they are implemented in SoNIC's software component. SoNIC provides full access to the PHY in software; as a result, all of the functionality in the PHY that manipulate bits (PCS2 and PCS3) are implemented in software.

For this partitioning between hardware and software, we chose an Altera Stratix IV FPGA [4] development board from HiTechGlobal [6] as our hardware platform. The board includes a PCIe Gen 2 interface (=32 Gbps) to the host PC, and is equipped with two SFP+ (Small Form-factor Pluggable) ports (Figure 2). The FPGA is equipped with 11.3 Gbps transceivers which can perform the 10 GbE PMA at line-speed. Once symbols are delivered to a transceiver on the FPGA they are converted to bits (PMA), and then transmitted to the host via PCIe by direct memory access (DMA). This board satisfies all the requirements discussed in the previous Section 2.2.

3.2 Realtime Capability

To achieve realtime, it is important to reduce any synchronization overheads between hardware and software, and between multiple pipelined cores. In SoNIC, the hardware does not generate interrupts when receiving or transmitting. Instead, the software decides when to initiate a DMA transaction by *polling* a value from a shared data memory structure where only the hardware writes. This approach is called *pointer polling* and is better than interrupts because there is always data to transfer due to the nature of continuous bitstreams in 10 GbE.

In order to synchronize multiple pipelined cores, a *chasing-pointer FIFO* from Sora [33] is used which supports low-latency pipelining. The FIFO removes the need for a shared synchronization variable and instead uses a flag to indicate whether a FIFO entry is available to reduce the synchronization overheads. In our implementation, we improved the FIFO by avoiding memory operations as well. Memory allocation and page faults are expensive and must be avoided to meet the realtime capability. Therefore, each FIFO entry in SoNIC is preallocated during initialization. In addition, the number of entries in a FIFO is kept small so that the amount of memory required for a port can fit into the shared L3 cache.

We use the Intel Westmere processor to achieve high performance. Intel Westmere is a Non-Uniform Memory Access (NUMA) architecture that is efficient for implementing packet processing applications [14, 18, 28, 30]. It is further enhanced by a new instruction `PCLMULQDQ` which was recently introduced. This instruction performs carry-less multiplication and we use it to implement a fast CRC algorithm [16] that the MAC requires. Using `PCLMULQDQ` instruction makes it possible to implement a CRC engine that can process 10 GbE bits at line-speed on a single core.

3.3 Scalability and Efficiency

The FPGA board we use is equipped with two physical 10 GbE ports and a PCIe interface that can support up to 32 Gbps. Our design goal is to support two physical ports per board. Consequently, the number of CPU cores and the amount of memory required for one port must be bounded. Further, considering the intense computation required for the PCS and MAC, and that recent processors come with four to six or even eight cores per socket, our goal is to limit the number of CPU cores required per port to the number of cores available in a socket. As a result, for one port we implement four dedicated kernel threads each running on different CPU cores. We use a PCS thread and a MAC thread on both the transmit and receive paths. We call our threads: TX PCS, RX PCS, TX MAC and RX MAC. Interrupt requests (IRQ) are re-

routed to unused cores so that SoNIC threads do not give up the CPU and can meet the realtime requirements.

Additionally, we use memory very efficiently: DMA buffers are preallocated and reused and data structures are kept small to fit in the shared L3 cache. Further, by utilizing memory efficiently, dedicating threads to cores, and using multi-processor QPI support, we can linearly increase the number of ports with the number of processors. QPI provides enough bandwidth to transfer data between sockets at a very fast data rate ($>$ 100 Gbps).

A significant design issue still abounds: communication and CPU core utilization. The way we pipeline CPUs, i.e. sharing FIFOs depends on the application. In particular, we pipeline CPUs differently depending on the application to reduce the number of active CPUs; unnecessary CPUs are returned to OS. Further, we can enhance communication with a general rule of thumb: take advantage of the NUMA architecture and L3 cache and place closely related threads on the same CPU socket.

Figure 3 illustrates examples of how to share FIFOs among CPUs. An arrow is a shared FIFO. For example, a packet generator only requires TX elements (Figure 3a); RX PCS simply receives and discards bitstreams, which is required to keep a link active. On the contrary, a packet capturer requires RX elements (Figure 3b) to receive and capture packets. TX PCS is required to establish and maintain a link to the other end by sending `/I/`s. To create a network profiling application, both the packet generator and packet capturer can run on different sockets simultaneously.

3.4 Precision

As discussed in Section 3.1, the PCS2 and PCS3 are implemented in software. Consequently, the software receives the entire raw bitstream from the hardware. While performing PCS2 and PCS3 functionalities, a PCS thread records the number of bits in between and within each Ethernet frame. This information can later be retrieved by a user application. Moreover, SoNIC allows users to precisely control the number of bits in between frames when transmitting packets, and can even change the value of any bits. For example, we use this capability to give users fine-grain control over packet generators and can even create virtually undetectable covert channels.

3.5 User Interface

SoNIC exposes fine-grained control over the path that a bitstream travels in software. SoNIC uses the `ioctl` system call for control, and the character device interface to transfer information when a user application needs to retrieve data. Moreover, users can assign which CPU cores or socket each thread runs on to optimize the path.

To allow further flexibility, SoNIC allows additional application-specific threads, called APP threads, to be pipelined with other threads. A character device is used to communicate with these APP threads from userspace. For instance, users can implement a logging thread pipelined with receive path threads (RX PCS and/or RX MAC). Then the APP thread can deliver packet information along with precise timing information to userspace via a character device interface. There are two constraints that an APP thread must always meet: Performance and pipelining. First, whatever functionality is implemented in an APP thread, it must be able to perform it faster than 10.3125 Gbps for any given packet stream in order to meet the realtime capability. Second, an APP thread must be properly pipelined with other threads, i.e. input/output FIFO must be properly set. Currently, SoNIC supports one APP thread per port.

```
1: #include "sonic.h"
2:
3: struct sonic_pkt_gen_info info = {
4:     .pkt_num   = 1000000000UL,
5:     .pkt_len   = 1518,
6:     .mac_src   = "00:11:22:33:44:55",
7:     .mac_dst   = "aa:bb:cc:dd:ee:ff",
8:     .ip_src    = "192.168.0.1",
9:     .ip_dst    = "192.168.0.2",
10:    .port_src  = 5000,
11:    .port_dst  = 5000,
12:    .idle   = 12, };
13:
14: fd1 = open(SONIC_CONTROL_PATH, O_RDWR);
15: fd2 = open(SONIC_PORT1_PATH, O_RDONLY);
16:
17: ioctl(fd1, SONIC_IOC_RESET)
18: ioctl(fd1, SONIC_IOC_SET_MODE, SONIC_PKT_GEN_CAP)
19: ioctl(fd1, SONIC_IOC_PORT0_INFO_SET, &info)
20: ioctl(fd1, SONIC_IOC_RUN, 10)
21:
22: while ((ret = read(fd2, buf, 65536)) > 0) {
23:  // process data }
24:
25: close(fd1);
26: close(fd2);
```

Figure 4: E.g. SoNIC Packet Generator and Capturer

Figure 4 illustrates the source code of an example use of SoNIC as a packet generator and capturer. After `SONIC_IOC_SET_MODE` is called (line 18), threads are pipelined as illustrated in Figure 3a and 3b. After `SONIC_IOC_RUN` command (line 20), port 0 starts generating packets given the information from `info` (line 3-12) for 10 seconds (line 20) while port 1 starts capturing packets with very precise timing information. Captured information is retrieved with `read` system calls (line 22-23) via a character device. As a packet generator, users can set the desired number of `/I/`s between packets (line 12). For example, twelve `/I/` characters will achieve the maximum data rate. Increasing the number of `/I/` characters will decrease the data rate.

3.6 Discussion

We have implemented SoNIC to achieve the design goals described above, namely, software access to the PHY, realtime capability, scalability, high precision, and an interactive user interface. Figure 5 shows the major components of our implementation. From top to bottom, user

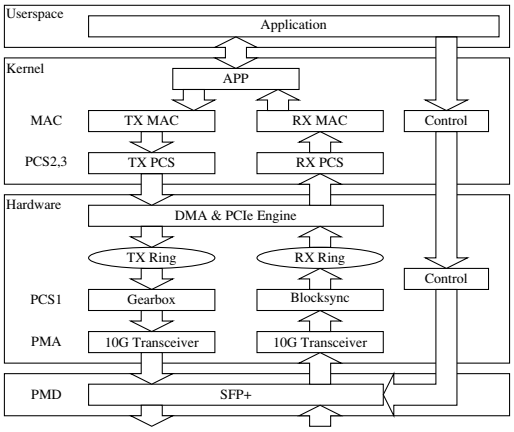


Figure 5: SoNIC architecture

applications, software as a loadable Linux kernel module, hardware as a firmware in FPGA, and a SFP+ optical transceiver. Although Figure 5 only illustrates one physical port, there are two physical ports available in SoNIC. SoNIC software consists of about 6k lines of kernel module code, and SoNIC hardware consists of 6k lines of Verilog code excluding auto-generated source code by Altera Quartus [3] with which we developed SoNIC's hardware modules.

The idea of accessing the PHY in software can be applied to other physical layers with different speeds. The 1 GbE and 40 GbE PHYs are similar to the 10 GbE PHY in that they run in full duplex mode, and maintain continuous bitstreams. Especially, the 40GbE PCS employes four PCS lanes that implements 64B/66B encoding as in the 10GbE PHY. Therefore, it is possible to access the PHYs of them with appropriate clock cycles and hardware supports. However, it might not be possible to implement four times faster scrambler with current CPUs.

In the following sections, we will highlight how SoNIC's implementation is optimized to achieve high performance, flexibility, and precision.

4 Optimizations

Performance is paramount for SoNIC to achieve its goals and allow software access to the entire network stack. In this section we discuss the software (Section 4.1) and hardware (Section 4.2) optimizations that we employ to enable SoNIC. Further, we evaluate each optimization (Sections 4.1 and 4.2) and demonstrate that they help to enable SoNIC and network research applications (Section 5) with high performance.

4.1 Software Optimizations

MAC Thread Optimizations As stated in Section 3.2, we use `PCLMULQDQ` instruction which performs carryless multiplication of two 64-bit quadwords [17] to implement the fast CRC algorithm [16]. The algorithm *folds* a large chunk of data into a smaller chunk using the `PCLMULQDQ` instruction to efficiently reduce the size of data. We adapted this algorithm and implemented it using inline assembly with optimizations for small packets.

PCS Thread Optimizations Considering there are 156 million 66-bit blocks a second, the PCS must process each block in less than 6.4 nanoseconds. Our optimized (de-)scrambler can process each block in 3.06 nanoseconds which even gives enough time to implement decode/encode and DMA transactions within a single thread.

In particular, the PCS thread needs to implement the (de-)scrambler function, $G(x) = 1 + x^{39} + x^{58}$, to ensure that a mix of 1's and 0's are always sent (DC balance). The (de-)scrambler function can be implemented with Algorithm 1, which is very computationally expensive [15] taking 320 shift and 128 xor operations (5 shift operations and 2 xors per iteration times 64 iterations). In fact, our original implementation of Algorithm 1 performed at 436 Mbps, which was not sufficient and became the bottleneck for the PCS thread. We optimized and reduced the scrambler algorithm to a *total* of 4 shift and 4 xor operations (Algorithm 2) by carefully examining how hardware implements the scrambler function [34]. Both Algorithm 1 and 2 are equivalent, but Algorithm 2 runs 50 times faster (around 21 Gbps).

Algorithm 1 Scrambler

$s \leftarrow$ state
$d \leftarrow$ data
for $i = 0 \rightarrow 63$ **do**
  $in \leftarrow (d >> i)$ & 1
  $out \leftarrow (in \oplus (s >> 38) \oplus (s >> 57))$ & 1
  $s \leftarrow (s << 1) \mid out$
  $r \leftarrow r \mid (out << i)$
  state $\leftarrow s$
end for

Algorithm 2 Parallel Scrambler

$s \leftarrow$ state
$d \leftarrow$ data
$r \leftarrow (s >> 6) \oplus (s >> 25) \oplus d$
$r \leftarrow r \oplus (r << 39) \oplus (r << 58)$
state $\leftarrow r$

Memory Optimizations We use *packing* to further improve performance. Instead of maintaining an array of data structures that each contains metadata and a pointer to the packet payload, we pack as much data as possible into a preallocated memory space: Each packet structure contains metadata, packet payload, and an offset to the next packet structure in the buffer. This packing helps to reduce the number of page faults, and allows SoNIC to process small packets faster. Further, to reap the benefits of the `PCLMULQDQ` instruction, the first byte of each packet is always 16-byte aligned.

Evaluation We evaluated the performance of the TX MAC thread when computing CRC values to assess the performance of the fast CRC algorithm and packing packets we implemented relative to batching an array of

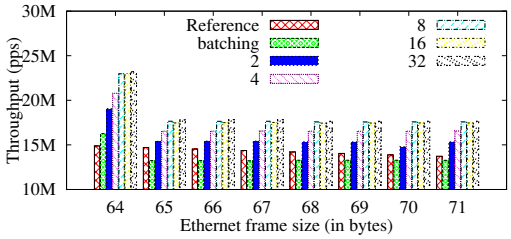


Figure 6: Throughput of packing

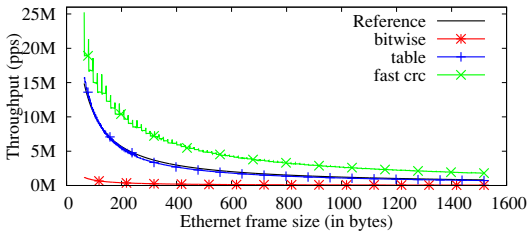


Figure 7: Throughput of different CRC algorithms

packets. For comparison, we computed the theoretical maximum throughput (Reference throughput) in packets per second (pps) for any given packet length (i.e. the pps necessary to achieve the maximum throughput of 10 Gbps less any protocol overhead).

If only one packet is packed in the buffer, packing will perform the same as batching since the two are essentially the same in this case. We doubled the factor of packing from 1 to 32 and assessed the performance of packing each time, i.e. we doubled the number of packets written to a single buffer. Figure 6 shows that packing by a factor of 2 or more always outperforms the Reference throughput and is able to achieve the max throughput for small packets while batching does not.

Next, we compared our fast CRC algorithm against two CRC algorithms that the Linux Kernel provides. One of the Linux CRC algorithms is a naive bit computation and the other is a table lookup algorithm. Figure 7 illustrates the results of our comparisons. The x-axis is the length of packets tested while the y-axis is the throughput. The Reference line represents the maximum possible throughput given the 10 GbE standard. Packet lengths range the spectrum of sizes allowed by 10 GbE standard from 64 bytes to 1518 bytes. For this evaluations, we allocated 16 pages *packed* with packets of the same length and computed CRC values with different algorithms for 1 second. As we can see from Figure 7, the throughput of the table lookup closely follows the Reference line; however, for several packet lengths, it underperforms the Reference line and is unable to achieve the maximum throughput. The fast CRC algorithm, on the other hand, outperforms the Reference line and target throughput for all packet sizes.

Lastly, we evaluated the performance of pipelining and using multiple threads on the TX and RX paths. We tested a full path of SoNIC to assess the performance as packets travel from the TX MAC to the TX PCS for transmission and up the reverse path for receiving from the RX PCS to the RX MAC and to the APP (as a logging thread). We do not show the graph due to space constraints, but all threads perform better than the Reference target throughput. The overhead of FIFO is negligible when we compare the throughputs of individual threads to the throughput when all threads are pipelined together. Moreover, when using two ports simultaneously (two full instances of receive and transmit SoNIC paths), the throughput for both ports achieve the Reference target maximum throughput.

4.2 Hardware Optimizations

DMA Controller Optimizations Given our desire to transfer large amounts of data (more than 20 Gbps) over the PCIe, we implemented a high performance DMA controller. There are two key factors that influenced our design of the DMA controller. First, because the incoming bitstream is a continuous 10.3125 Gbps, there must be enough buffering inside the FPGA to compensate for a transfer latency. Our implementation allocates four rings in the FPGA for two ports (Figure 5 shows two of the rings for one port). The maximum size of each ring is 256 KB, with the size being limited by the amount of SRAM available.

The second key factor we needed to consider was the efficient utilization of bus bandwidth. The DMA controller operates at a data width of 128 bits. If we send a 66-bit data block over the 128-bit bus every clock cycle, we will waste 49% of the bandwidth, which was not acceptable. To achieve more efficient use of the bus, we create a `sonic_dma_page` data structure and separated the syncheader from the packet payload before storing a 66-bit block in the data structure. Sixteen two-bit syncheaders are concatenated together to create a 32-bit integer and stored in the `syncheaders` field of the data structure. The 64-bit packet payloads associated with these syncheaders are stored in the `payloads` field of the data structure. For example, the ith 66-bit PCS block from a DMA page consists of the two-bit sync header from `syncheaders`$[i/16]$ and the 64-bit payload from `payloads`$[i]$. With this data structure there is a 32-bit overhead for every page, however it does not impact the overall performance.

PCI Express Engine Optimizations When SoNIC was first designed, it only supported a single port. As we scaled SoNIC to support multiple ports simultaneously, the need for multiplexing traffic among ports over the single PCIe link became a significant issue. To solve this issue, we employ a two-level arbitration scheme to provide fair arbitration among ports. A lower level arbiter is a fixed-priority arbiter that works within a sin-

Configuration	Same Socket?	# pages	Throughput (RX)		# pages	Throughput (TX)		Realtime?
			Port 0	Port 1		Port 0	Port 1	
Single RX		16	25.7851					
Dual RX	Yes	16	13.9339	13.899				
	No	8	14.2215	13.134				
Single TX					16	23.7437		
Dual TX	Yes				16	14.0082	14.048	
	No				16	13.8211	13.8389	
Single RX/TX		16	21.0448		16	22.8166		
Dual RX/TX	Yes	4	10.7486	10.8011	8	10.6344	10.7171	No
		4	11.2392	11.2381	16	12.384	12.408	Yes
		8	13.9144	13.9483	8	9.1895	9.1439	Yes
		8	14.1109	14.1107	16	10.6715	10.6731	Yes
	No	4	10.5976	10.183	8	10.3703	10.1866	No
		4	10.9155	10.231	16	12.1131	11.7583	Yes
		8	13.4345	13.1123	8	8.3939	8.8432	Yes
		8	13.4781	13.3387	16	9.6137	10.952	Yes

Table 1: DMA throughput. The numbers are average over eight runs. The delta in measurements was within 1% or less.

gle port and arbitrates between four basic Transaction Level Packet (TLP) types: Memory, I/O, configuration, and message. The TLPs are assigned with fixed priority in favor of the write transaction towards the host. The second level arbiter implements a virtual channel, where the Traffic Class (TC) field of TLP's are used as demultiplexing keys. We implemented our own virtual channel mechanism in SoNIC instead of using the one available in the PCIe stack since virtual channel support is an optional feature for vendors to comply with. In fact, most chipsets on the market do not support the virtual channel mechanism. By implementing the virtual channel support in SoNIC, we achieve better portability since we do not rely on chip vendors that enable PCI arbitration.

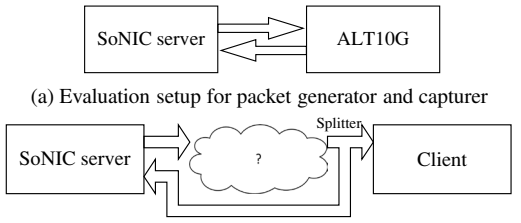
Evaluation We examined the maximum throughput for DMA between SoNIC hardware and SoNIC software to evaluate our hardware optimizations. It is important that the bidirectional data rate of each port of SoNIC is greater than 10.3125 Gbps. For this evaluation, we created a DMA descriptor table with one entry, and changed the size of memory for each DMA transaction from one page (4K) to sixteen pages (64KB), doubling the number of pages each time. We evaluated the throughput of a single RX or TX transaction, dual RX or TX transactions, and full bidirectional RX and TX transactions with both one and two ports (see the rows of Table 1). We also measured the throughput when traffic was sent to one or two CPU sockets.

Table 1 shows the DMA throughputs of the transactions described above. We first measured the DMA without using pointer polling (see Section 3.2) to obtain the maximum throughputs of the DMA module. For single RX and TX transactions, the maximum throughput is close to 25 Gbps. This is less than the theoretical maximum throughput of 29.6 Gbps for the x8 PCIe interface, but closely matches the reported maximum throughput of 27.5 Gbps [2] from Altera design. Dual RX or TX transactions also resulted in throughputs similar to the reference throughputs of Altera design.

Next, we measured the full bidirectional DMA transactions for both ports varying the number of pages again. As shown in the bottom half of Table 1, we have multiple configurations that support throughputs greater than 10.3125 Gbps for full bidirections. However, there are a few configurations in which the TX throughput is less than 10.3125 Gbps. That is because the TX direction requires a small fraction of RX bandwidth to fetch the DMA descriptor. If RX runs at maximum throughput, there is little room for the TX descriptor request to get through. However, as the last column on the right indicates these configurations are still able to support the realtime capability, i.e. consistently running at 10.3125 Gbps, when *pointer polling* is enabled. This is because the RX direction only needs to run at 10.3125 Gbps, less than the theoretical maximum throughput (14.8 Gbps), and thus gives more room to TX. On the other hand, two configurations where both RX and TX run faster than 10.3125 Gbps for full bidirection are not able to support the realtime capability. For the rest of the paper, we use 8 pages for RX DMA and 16 pages for TX DMA.

5 Network Research Applications

How can SoNIC enable flexible, precise and novel network research applications? Specifically, what unique value does software access to the PHY buy? As discussed in Section 2.5, SoNIC can literally count the number of bits between and within packets, which can be used for timestamping at the sub-nanosecond granularity (again each bit is 97 ps wide, or about $\sim$0.1 ns). At the same time, access to the PHY allows users control over the number of `idles` (`/I/`s) between packets when generating packets. This fine-grain control over the `/I/`s means we can precisely control the data rate and the distribution of interpacket gaps. For example, the data rate of a 64B packet stream with uniform 168 `/I/`s is 3 Gbps. When this precise packet generation is combined with exact packet capture, also enabled by SoNIC, we can improve the accuracy of any research based on interpacket delays [11, 19, 22, 23, 24, 25, 26, 32, 35, 37].



(a) Evaluation setup for packet generator and capturer

(b) Evaluation setup for other applications

Figure 8: Experiment setups for SoNIC

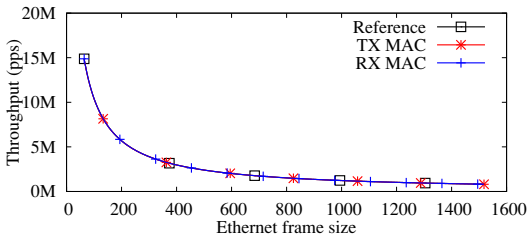


Figure 9: Throughput of packet generator and capturer

In this section, we demonstrate that SoNIC can precisely and flexibly characterize and profile commodity network components like routers, switches, and NICs. Section 5.4 discusses the profiling capability enabled by SoNIC. Further, in Section 5.5, we demonstrate that SoNIC can be used to create a covert timing channel that is not detectable by applications that do not have access to the PHY and data link layers and that do not have accurate timestamping capabilities. First, however, we demonstrate SoNIC's accurate packet generation capability in Section 5.2 and packet capture capability in Section 5.3, which are unique contributions and can enable unique network research in and of themselves given both the flexibility, control, and precision.

Interpacket delay (IPD) and *interpacket gap* (IPG) are defined as follows: IPD is the time difference between the first bit of successive packets, while IPG is the time difference between the last bit of the first packet and the first bit of the next packet.

5.1 Experiment Setup

We deployed SoNIC on a Dell Precision T7500 workstation. This workstation is a dual socket, 2.93 GHz six core Xeon X5670 (Westmere) with 12 MB of shared L3 cache and 12 GB of RAM, 6 GB connected to each of the two CPU sockets. The machine has two PCIe Gen 2.0 x8 slots, where SoNIC hardware is plugged in, and is equipped with an Intel 5520 chipset connected to each CPU socket by a 6.4 GT/s QuickPath Interconnect (QPI). Two Myricom 10G-SFP-LR transceivers are plugged into SoNIC hardware. We call this machine the SoNIC server. For our evaluations we also deployed an Altera 10G Ethernet design [1] (we call this ALT10G) on an FPGA development board. This FPGA is the same type as the one SoNIC uses and is deployed on a server identical to the SoNIC server. We also used a server identical to the SoNIC server with a Myricom 10G-PCIE2-8B2-2S dual 10G port NIC (we call this Client).

To evaluate SoNIC as a packet generator and capturer, we connected the SoNIC board and the ALT10G board directly via optic fibers (Figure 8a). ALT10G allows us to generate random packets of any length and with the minimum IPG to SoNIC. It also provides us with detailed statistics such as the number of valid/invalid Ethernet frames, and frames with CRC errors. We used this feature to stress test SoNIC for the packet generator and capturer. We compared these numbers from ALT10G with statistics from SoNIC to verify the correctness of SoNIC.

To evaluate other applications, we used port 0 of SoNIC server to generate packets to the Client server via an arbitrary network, and split the signal with a fiber optic splitter so that the same stream can be directed to both the Client and port 1 of the SoNIC server performing the packet capture (Figure 8b). We used various network topologies composed of Cisco 4948, and IBM BNT G8264 switches for the network between the SoNIC server and the Client.

5.2 Packet Generator

Packet generation is important for network research. It can stress test end-hosts, switches/routers, or a network itself. Moreover, packet generation can be used for replaying a trace, studying distributed denial of service (DDoS) attacks, or probing firewalls.

In order to claim that a packet generator is accurate, packets need to be crafted with fine-grained precision (minimum deviations in IPD) at the maximum data rate. However, this fine-grained control is not usually exposed to users. Further, commodity servers equipped with a commodity NIC often does not handle small packets efficiently and require batching [14, 18, 28, 30]. Thus, the sending capability of servers/software-routers are determined by the network interface devices. Myricom Sniffer 10G [8] provides line-rate packet injection capability, but does not provide fine-grained control of IPGs. Hardware based packet generators such as ALT10G can precisely control IPGs, but do not provide any interface for users to flexibly control them.

We evaluated SoNIC as a packet generator (Figure 3a). Figure 10 compares the performance of SoNIC to that of Sniffer 10G. Note, we do not include ALT10G in this evaluation since we could not control the IPG to generate packets at 9 Gbps. We used two servers with Sniffer 10G enabled devices to generate 1518B packets at 9 Gbps between them. We split the stream so that SoNIC can capture the packet stream in the middle (we describe this capture capability in the following section). As the graph shows, Sniffer 10G allows users to generate packets at desired data rate, however, it does not give the con-

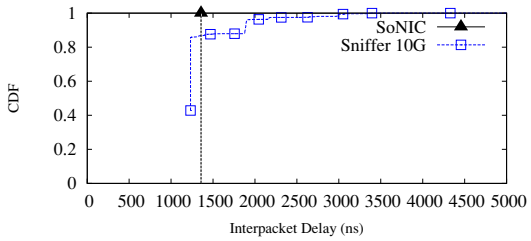


Figure 10: Comparison of packet generation at 9 Gbps

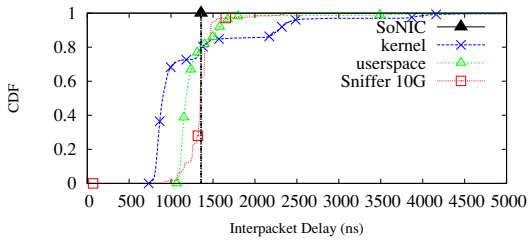


Figure 11: Comparison of timestamping

trol over the IPD; that is, 85.65% packets were sent in a burst (instantaneous 9.8 Gbps and minimum IPG (14 /I/s)). SoNIC, on the other hand, can generate packets with uniform distribution. In particular, SoNIC generated packets with no variance for the IPD (i.e. a single point on the CDF, represented as a triangle). Moreover, the maximum throughput perfectly matches the Reference throughput (Figure 9) while the TX PCS consistently runs at 10.3125 Gbps (which is not shown). In addition, we observed no packet loss, bit errors, or CRC errors during our experiments.

SoNIC packet generator can easily achieve the maximum data rate, and allows users to precisely control the number of /I/s to set the data rate of a packet stream. Moreover, with SoNIC, it is possible to inject less /I/s than the standard. For example, we can achieve 9 Gbps with 64B packets by inserting only eight /I/s between packets. This capability is not possible with any other (software) platform. In addition, if the APP thread is carefully designed, users can flexibly inject a random number of /I/s between packets, or the number of /I/s from captured data. SoNIC packet generator is thus by far the most flexible and highest performing.

5.3 Packet Capturer

A packet capturer (a.k.a. packet sniffer, or packet analyzer) plays an important role in network research; it is the opposite side of the same coin as a packet generator. It can record and log traffic over a network which can later be analyzed to improve the performance and security of networks. In addition, capturing packets with precise timestamping is important for High Frequency Trading [21, 31] or latency sensitive applications.

Similar to the sending capability, the receiving capability of servers and software routers is inherently limited by the network adapters they use; it has been shown that some NICs are not able to receive packets at line speed for certain packet sizes [30]. Furthermore, if batching is used, timestamping is significantly perturbed if done in kernel or userspace [15]. High-performance devices such as Myricom Sniffer10G [8, 20] provide the ability of sustained capture of 10 GbE by bypassing kernel network stack. It also provides timestamping at 500 ns resolution for captured packets. SoNIC, on the other hand, can receive packets of any length at line-speed with precise timestamping. For instance, we will show in Section 5.5 that we can create a covert timing channel that is undetectable to a Sniffer 10G enabled system or any other software-enabled systems[14, 18, 28, 30].

Putting it all together, when we use SoNIC as a packet capturer (Figure 3b), we are able to receive at the full Reference data rate (Figure 9). For the APP thread, we implemented a simple logging application which captures the first 48 bytes of each packet along with the number of /I/s and bits between packets. Because of the relatively slow speed of disk writes, we store the captured information in memory. This requires about 900MB to capture a stream of 64 byte packets for 1 second, and 50 MB for 1518 byte packets. We use ALT10G to generate packets for 1 second and compare the number of packets received by SoNIC to the number of packets generated.

SoNIC has perfect packet capture capabilities with flexible control in software. In particular, Figure 11 shows that given a 9 Gbps generated traffic with uniform IPD (average IPD=1357.224ns, stdev=0), SoNIC captures what was sent; this is shown as a single triangle at (1357.224, 1). All the other packet capture methods within userspace, kernel or a mixture of hardware timestamping in userspace (Sniffer 10G) failed to accurately capture what was sent. We receive similar results at lower bandwidths as well.

5.4 Profiler

Interpacket delays are a common metric for network research. It can be used to estimate available bandwidth [23, 24, 32], increase TCP throughput [37], characterize network traffic [19, 22, 35], and detect and prevent covert timing channels [11, 25, 26]. There are a lot of metrics based on IPD for these areas. We argue that SoNIC can increase the accuracy of those applications because of its precise control and capture of IPDs. In particular, when the SoNIC packet generator and capturer are combined, i.e. one port transmits packets while the other port captures, SoNIC can be a flexible platform for various studies. As an example, we demonstrate how SoNIC can be used to profile network switches.

Switches can be generally divided into two categories: store-and-forward and cut-through switches. Store-and-forward switches decode incoming packets, buffers them before making a routing decision. On the other hand, cut-

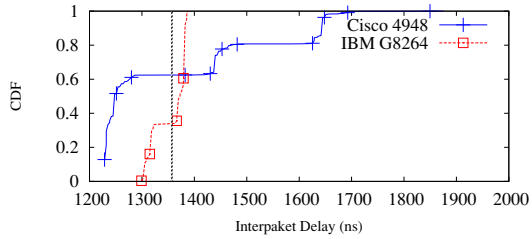


Figure 12: IPDs of Cisco 4948 and IBM G8264. 1518B packets at 9 Gbps

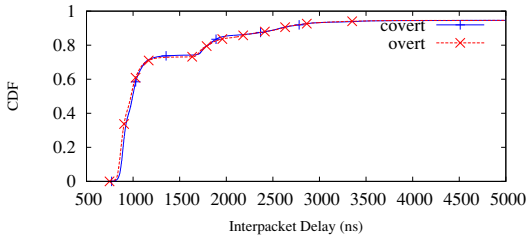


Figure 13: Packet distribution of covert channel and overt channel where $\Delta = 3562$ and $\varepsilon = 128$

through switches route incoming packets before entire packets are decoded to reduce the routing latency. We generated 1518B packets with uniform 1357.19 ns IPD (=9 Gbps) to a Cisco 4948 (store-and-forward) switch and a IBM BNT G8264 (cut-through) switch. These switches show different characteristics as shown in Figure 12. The x-axis is the interpacket delay; the y-axis is the cumulative distribution function. The long dashed vertical line on the left is the original IPD injected to the packet stream.

There are several takeaways from this experiment. First, the IPD for generated packets had no variance; none. The generated IPD produced by SoNIC was *always* the same. Second, the cut-through switch introduces IPD variance (stdev=31.6413), but less than the IPD on the store-and-forward switch (stdev=161.669). Finally, the average IPD was the same for both switches since the data rate was the same: 1356.82 (cut-through) and 1356.83 (store-and-forward). This style of experiment can be used to profile and fingerprint network components as different models show different packet distributions.

5.5 Covert Channels

Covert channels in a network is a side channel that can convey a hidden message embedded to legitimate packets. There are two types of covert channels: Storage and timing channels. Storage channels use a specific location of a packet to deliver a hidden message. Timing channels modulate resources over time to deliver a message [38]. Software access to the PHY opens the possibility for both storage and timing channels. The ability to detect covert channels is important because a rogue router or an end-host can create covert channels to deliver sensitive information without alerting network administrators. We will discuss how to create covert channels with SoNIC, and thus argue that SoNIC can be used to detect any potentially undetectable covert channels in local area network or data center networks.

When SoNIC enabled devices are directly connected between two end-hosts, a secret message can be communicated without alerting the end-host. In particular, there are unused bits in the 10 GbE standard where an adversary can inject bits to create covert messages. Unfortunately, such a covert storage channel can only work for one hop. On the other hand, precisely controlling IPG can create a timing channel that can travel multiple hops in the network, and that cannot be easily detected with inaccurate timestamping. Such a covert timing channel is based on the two observations: First, timing channel detection is usually performed in the application layer, and depends on the inherently inaccurate timestamping from kernel or userspace. Secondly, switches perturb the IPD, although the difference is still bounded, i.e. an IPD does not increase to an arbitrarily long one. Therefore, if we can modulate IPGs in a way that allows switches to preserve the gaps while making them indistinguishable to kernel/userspace timestamping, it is possible to create a virtually undetectable timing channel from applications operating at layers higher than layer 2.

We experimented with the creation of a simple timing channel. Let Δ be an uniform IPG for a packet stream. Then a small time window can be used to signal 1's and 0's. For example, IPG with $\Delta - \varepsilon$ /I/s represents 0 (if $\Delta - \varepsilon < 12$, we set it to 12, the minimum IPG) and $\Delta + \varepsilon$ /I/s represents 1. If the number of 1' and 0's meets the DC balance, the overall data rate will be similar to a packet stream with uniform IPGs of Δ /I/s. To demonstrate the feasibility of this approach, we created a network topology such that a packet travels from SoNIC through a Cisco 4948, IBM G8264, a different Cisco 4948, and then to the SoNIC server and the Client server with a fiber splitter (Figure 8b). SoNIC generates 1518B packets with $\Delta = 170,\ 1018,\ 3562,\ 13738$ (= 9, 6, 3, 1 Gbps respectively), with $\varepsilon = 16, 32, \ldots, 2048$. Then, we measured the bit error ratio (BER) with captured packets. Table 2 summarizes the result. We only illustrated the smallest ε from each Δ that achieves BER less than 1%. The takeaway is that by modulating IPGs at 100 ns scale, we can create a timing channel.

Data rate (Gbps)	Δ	δ (# /I/s)	δ (in ns)	BER
9	170	2048	1638.9	0.0359
6	1018	1024	819.4	0.0001
3	3562	128	102.4	0.0037
1	13738	128	102.4	0.0035

Table 2: Bit error ratio of timing channels

Figure 13 illustrates the IPDs with kernel timestamping from overt channel and covert channel when $\Delta =$

3562 and $\varepsilon = 128$. The two lines closely overlap, indicating that it is not easy to detect. There are other metrics to evaluate the undetectability of a timing channel [11, 25, 26], however they are out of the scope of this paper, and we do not discuss them.

6 Related Works

6.1 Reconfigurable Network Hardware

Reconfigurable network hardware allows for the experimentation of novel network system architectures. Previous studies on reconfigurable NICs [36] showed that it is useful for exploring new I/O virtualization technique in VMMs. NetFPGA [27] allows users to experiment with FPGA-based router and switches for research in new network protocols and intrusion detection [10, 12, 29, 39]. The recent NetFPGA 10G platform is similar to the SoNIC platform. While NetFPGA 10G allows user to access the layer 2 and above, SoNIC allows user to access the PHY. This means that user can access the entire network stack in software using SoNIC.

6.2 Timestamp

The importance of timestamping has long been established in the network measurement community. Prior work either does not provide precise enough timestamping, or requires special devices. Packet stamping in userspace or kernel suffers from the imprecision introduced by the OS layer [13]. Timestamping in hardware either requires offloading the network stack to a custom processor [37], or relies on an external clock source [5], which makes the device hard to program and inconvenient to use in a data center environment. Data acquisition and generation (DAG) cards [5] additionally offer globally synchronized clocks among multiple devices, whereas SoNIC only supports delta timestamping.

Although BiFocals [15] is able to provide an exact timestamping, the current state-of-art has limitations that prevented it from being a portable and realtime tool. BiFocals can store and analyze only a few milliseconds worth of a bitstream at a time due to the small memory of the oscilloscope. Furthermore, it requires thousands of CPU hours for converting raw optic waveforms to packets. Lastly, the physics equipment used by BiFocals are expensive and not easily portable. Its limitations motivated us to design SoNIC to achieve the realtime exact precision timestamping.

6.3 Software Defined Radio

The Software Defined Radio (SDR) allows easy, rapid prototyping of wireless network in software. Open-access platforms such as the Rice University's WARP [9] allow researchers to program both the physical and network layer on a single platform. Sora [33] presented the first SDFR platform that fully implements IEEE 802.11b/g on a commodity PC. AirFPGA [39] implemented a SDR platform on NetFPGA, focusing on building a chain of signal processing engines using commodity machines. SoNIC is similar to Sora in that it allows users to access and modify the PHY and MAC layers. The difference is that SoNIC must process multiple 10 Gbps channels which is much more computationally intensive than the data rate of wireless channels. Moreover, it is harder to synchronize hardware and software because a 10GbE link runs in a full duplex mode, unlike a wireless newtork.

6.4 Software Router

Although SoNIC is orthogonal to software routers, it is worth mentioning software routers because they share common techniques. SoNIC preallocates buffers to reduce memory overhead [18, 30], polls huge chunks of data from hardware to minimize interrupt overhead [14, 18], packs packets in a similar fashion to batching to improve performance [14, 18, 28, 30]. Software routers normally focus on scalability and hence exploit multi-core processors and multi-queue supports from NICs to distribute packets to different cores to process. On the other hand, SoNIC pipelines multiple CPUs to handle continuous bitstreams.

7 Conclusion

In this paper, we presented SoNIC which allows users to access the physical layer in realtime from software. SoNIC can generate, receive, manipulate and forward 10 GbE bitstreams at line-rate from software. Further, SoNIC gives systems programmers unprecedented precision for network measurements and research. At its heart, SoNIC utilizes commodity-off-the-shelf multi-core processors to implement part of the physical layer in software and employs an FPGA board to transmit optical signal over the wire. As a result, SoNIC allows cross-network-layer research explorations by systems programmers.

8 Availability

The SoNIC platform and source code is published under a BSD license and is freely available for download at `http://sonic.cs.cornell.edu`

9 Acknowledgements

This work was partially funded and supported by an IBM Faculty Award received by Hakim Weatherspoon, DARPA (No. D11AP00266), NSF CAREER (No. 1053757), NSF TRUST (No. 0424422), NSF FIA (No. 1040689), NSF CiC (No. 1047540), and an NSF EAGER (No. 1151268). We would like to thank our shepherd, Luigi Rizzo, and the anonymous reviewers for their comments. We especially thank Tudor Marian for his contributions and for the name, SoNIC, and Ethan Kao, Erluo Li, Jason Zhao, and Roman Averbukh for reviewing and comments.

References

[1] Altera. 10-Gbps Ethernet Reference Design. http://www.altera.com/literature/ug/10G_ethernet_user_guide.pdf.

[2] Altera. PCI Express High Performance Reference Design. http://www.altera.com/literature/an/an456.pdf.

[3] Altera Quartus II. http://www.altera.com/products/software/quartus-ii/subscription-edition.

[4] Altera Stratix IV FPGA. http://www.altera.com/products/devices/stratix-fpgas/stratix-iv/stxiv-index.jsp.

[5] Endace DAG Network Cards. http://www.endace.com/endace-dag-high-speed-packet-capture-cards.html.

[6] Hitechglobal. http://hitechglobal.com/Boards/Stratix4GX.html.

[7] IEEE Standard 802.3-2008. http://standards.ieee.org/about/get/802/802.3.html.

[8] Myricom Sniffer10G. http://www.myricom.com/sniffer.html.

[9] K. Amiri, Y. Sun, P. Murphy, C. Hunter, J. Cavallaro, and A. Sabharwal. WARP, a unified wireless network testbed for education and research. In *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, 2007.

[10] M. B. Anwer, M. Motiwala, M. b. Tariq, and N. Feamster. SwitchBlade: a platform for rapid deployment of network protocols on programmable hardware. In *Proceedings of the ACM SIGCOMM 2010 conference*, 2010.

[11] S. Cabuk, C. E. Brodley, and C. Shields. IP covert timing channels: Design and detection. In *Proceedings of the 11th ACM conference on Computer and communications security*, 2004.

[12] M. Casado. Reconfigurable networking hardware: A classroom tool. In *Proceedings of Hot Interconnects 13*, 2005.

[13] M. Crovella and B. Krishnamurthy. *Internet Measurement: Infrastructure, Traffic and Applications*. John Wiley and Sons, Inc, 2006.

[14] 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*, 2009.

[15] D. A. Freedman, T. Marian, J. H. Lee, K. Birman, H. Weatherspoon, and C. Xu. Exact temporal characterization of 10 Gbps optical wide-area network. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, 2010.

[16] V. Gopal, E. Ozturk, J. Guilford, G. Wolrich, W. Feghali, M. Dixon, and D. Karakoyunlu. Fast CRC computation for generic polynomials using PCLMULQDQ instruction. White paper, Intel, http://download.intel.com/design/intarch/papers/323102.pdf, December 2009.

[17] S. Gueron and M. E. Kounavis. Intel carry-less multiplication instruction and its usage for computing the GCM mode. White paper, Intel, http://software.intel.com/file/24918, January 2010.

[18] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference*, 2010.

[19] R. Jain, , and S. A. Routhier. Packet trains: Measurements and a new model for computer network traffic. *IEEE Journal On Selected Areas in Communications*, 4:986–995, 1986.

[20] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictble low latency for data center applications. In *Proceedings of the ACM Symposium on Cloud Computing*, 2012.

[21] R. R. Kompella, K. Levchenko, A. C. Snoeren, and G. Varghese. Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, 2009.

[22] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ACM Transaction on Networking*, 2(1), Feb. 1994.

[23] X. Liu, K. Ravindran, B. Liu, and D. Loguinov. Single-hop probing asymptotics in available bandwidth estimation: sample-path analysis. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, 2004.

[24] X. Liu, K. Ravindran, and D. Loguinov. Multi-hop probing asymptotics in available bandwidth estimation: stochastic analysis. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, 2005.

[25] Y. Liu, D. Ghosal, F. Armknecht, A.-R. Sadeghi, S. Schulz, and S. Katzenbeisser. Hide and seek in time: Robust covert timing channels. In *Proceedings of the 14th European conference on Research in computer security*, 2009.

[26] Y. Liu, D. Ghosal, F. Armknecht, A.-R. Sadeghi, S. Schulz, and S. Katzenbeisser. Robust and undetectable steganographic timing channels for i.i.d. traffic. In *Proceedings of the 12th international conference on Information hiding*, 2010.

[27] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA–an open platform for gigabit-rate network switching and routing. In *Proceedings of Microelectronics Systems Education*, 2007.

[28] T. Marian, K. S. Lee, and H. Weatherspoon. Netslices: Scalable multi-core packet processing in user-space. In *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2012.

[29] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown. Implementing an OpenFlow switch on the NetFPGA platform. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008.

[30] L. Rizzo. Netmap: a novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, 2012.

[31] D. Schneider. The Microsecond Market. *IEEE Spectrum*, 49(6):66–81, 2012.

[32] J. Strauss, D. Katabi, and F. Kaashoek. A measurement study of available bandwidth estimation tools. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, 2003.

[33] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G. M. Voelker. Sora: high performance software radio using general purpose multi-core processors. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, 2009.

[34] R. Walker, B. Amrutur, and T. Knotts. 64b/66b coding update. grouper.ieee.org/groups/802/3/ae/public/mar00/walker_1_0300.pdf.

[35] W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson. Self-similarity through high-variability: statistical analysis of Ethernet LAN traffic at the source level. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '95, 1995.

[36] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel. Concurrent direct network access for virtual machine monitors. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[37] T. Yoshino, Y. Sugawara, K. Inagami, J. Tamatsukuri, M. Inaba, and K. Hiraki. Performance optimization of TCP/IP over 10 gigabit Ethernet by precise instrumentation. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[38] S. Zander, G. Armitage, and P. Branch. A Survey of Covert Channels and Countermeasures in Computer Network Protocols. *Commun. Surveys Tuts.*, 9(3):44–57, July 2007.

[39] H. Zeng, J. W. Lockwood, G. A. Covington, and A. Tudor. AirFPGA: A software defined radio platform based on NetFPGA. In *NetFPGA Developers Workshop*, 2009.

Split/Merge: System Support for Elastic Execution in Virtual Middleboxes

Shriram Rajagopalan[†‡], Dan Williams[†], Hani Jamjoom[†], and Andrew Warfield[‡]

[†]IBM T. J. Watson Research Center, Yorktown Heights, NY
[‡]University of British Columbia, Vancouver, Canada

Abstract

Developing elastic applications should be easy. This paper takes a step toward the goal of generalizing elasticity by observing that a broadly deployed class of software—the network middlebox—is particularly well suited to dynamic scale. Middleboxes tend to achieve a clean separation between a small amount of per-flow network state and a large amount of complex application logic. We present a state-centric, systems-level abstraction for elastic middleboxes called *Split/Merge*. A virtual middlebox that has appropriately classified its state (e.g., per-flow state) can be dynamically scaled out (or in) by a Split/Merge system, but remains ignorant of the number of replicas in the system. Per-flow state may be transparently split between many replicas or merged back into one, while the network ensures flows are routed to the correct replica. As a result, Split/Merge enables load-balanced elasticity. We have implemented a Split/Merge system, called *FreeFlow*, and ported Bro, an open-source intrusion detection system, to run on it. In controlled experiments, FreeFlow enables a 25% reduction in maximum latency while eliminating hotspots during scale-out and a 50% quicker scale-in than standard approaches.

1 Introduction

The prevalence of Infrastructure as a Service (IaaS) clouds has given rise to a new breed of applications that better support *elasticity*: the ability to scale in or out to handle variations in workloads [17]. Fundamental to achieving elasticity is the ability to create or destroy virtual machine (VM) instances, or *replicas*, and partitioning work between them [14, 34]. For example, a 3-tier Web application may scale out the middle tier and balance requests between them. Consequently, the—virtual—middleboxes that these applications rely on (such as firewalls, intrusion detection systems, and protocol accelerators) must scale in a similar fashion. A recent survey of 57 enterprise networks of various sizes found that scalability was indeed critical for middleboxes [24].

Due to the diversity of cloud applications, supporting elasticity has been mostly the burden of the application or application-level framework [7]. For example, it is their responsibility to manage replicas and ensure that each replica will be assigned the same amount of work [1]. In the worst case, imbalances between replicas can result in inefficiencies, hotspots (e.g., overloaded replicas with degraded performance) or underutilized resources [33].

Unlike generic cloud applications, middleboxes share a unique property that can be exploited to achieve efficient, balanced elasticity. Despite the complex logic involved in routing or detecting intrusions, middleboxes are often implemented around the idea that each individual flow is an isolated context of execution [22, 26, 31]. Middleboxes typically classify packets to a specific flow, and then interact with data specific to that flow [9, 30]. By replicating a middlebox and adjusting the flows that each replica receives from the network—and the associated state held by each replica—any middlebox can maintain balanced load between replicas as the middlebox scales in or out.

To this end, we present a new hypervisor-level abstraction for virtual middleboxes called *Split/Merge*. A Split/Merge-aware middlebox may be replicated at will, yet remains oblivious to the existence of replicas. Split/Merge divides a middlebox application's state into two broad classes: *internal* and *external*. Internal state is treated similarly to application logic: it is required for a given replica to run, but is of no consequence outside that replica's execution. External state describes the application state that is actually scaled, and can be thought of as a large distributed data structure that is managed across all replicas. It can be further subdivided into to classes: *partitioned* and *coherent* state. Partitioned state is exclusively accessed, flow-specific data, and is the fun-

damental unit of reconfiguration in a Split/Merge system. Coherent state describes additional, often "global" state such as counters, that must remain consistent—either strongly or eventually—among all replicas.

We have designed and implemented *FreeFlow*, a system that implements Split/Merge to provide efficient, balanced elasticity for virtual middleboxes. FreeFlow splits flow-specific state among replicas and dynamically rebalances both existing and new flows across them. To enable middlebox applications to identify external state, associate it with network flows, and manage the migration of partitioned state between replicas, we have implemented an application-level *FreeFlow library*. In addition, we have implemented a *Split/Merge-aware software defined network* (SDN) that enables FreeFlow to partition the network such that each replica receives the appropriate network traffic even as partitioned state migrates between replicas.

FreeFlow enables elasticity by creating and destroying VM replicas, while balancing load between them. We have ported Bro [19], a real-world intrusion detection system, and built two synthetic middleboxes on FreeFlow. Using these middleboxes, we show that FreeFlow eliminates hotspots created during scale-out and inefficiencies during scale-in. In particular, it reduces the maximum latency by 25% after rebalancing flows during scale-out and achieves 50% quicker scale-in than standard approaches.

To summarize, the contributions of this paper are:

- a new hypervisor-level state abstraction that enables flow-related middlebox state to be identified, split, and merged between replica instances,
- a network abstraction that ensures that network input related to a particular flow-related piece of middlebox state arrives at the appropriate replica, and
- a system, FreeFlow, that implements Split/Merge alongside VM scale-in and scale-out to enable balanced elasticity for middleboxes.

The rest of this paper is organized as follows. Section 2 describes middleboxes and the Split/Merge abstraction. Section 3 describes the design and implementation of FreeFlow. Section 4 describes our experience in porting and building middleboxes for FreeFlow. Section 5 evaluates FreeFlow, Section 6 surveys related work, and Section 7 concludes.

2 Split/Merge

In this section, we describe the common structure in which middlebox state is organized. Motivated by this common structure, we define the three types of states exposed by the Split/Merge abstraction. We then describe how robust elasticity is achieved by tagging state and transparently partitioning network input across virtual middlebox replicas. We conclude the section with design challenges.

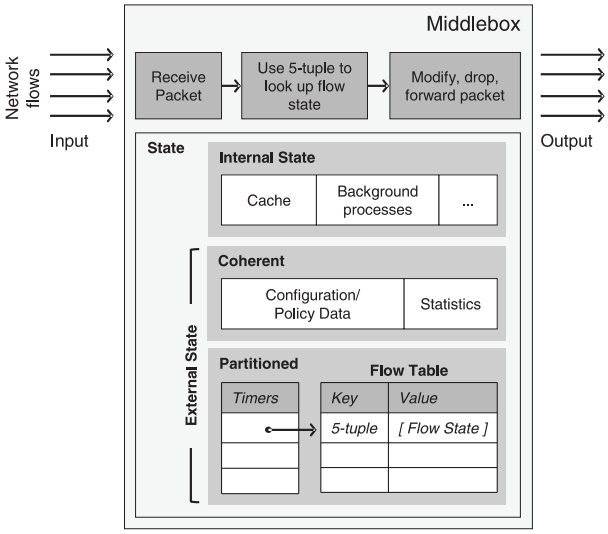


Figure 1: Typical structure of a middlebox

2.1 Anatomy of a Virtual Middlebox

A middlebox is defined as "any intermediary device performing functions other than the normal, standard functions of an IP router on the datagram path between a source host and destination host" [4]. Middleboxes can vary drastically in their function, performing such diverse tasks as network address translation, intrusion detection, packet filtering, protocol acceleration, and acting as a network proxy. However, middleboxes typically process packets and share the same basic structure [9,12,30].

Figure 1 shows the basic structure of a middlebox. State held by a middlebox can be characterized as policy and configuration data or as run-time responses to network flows [9, 26, 30, 31]. The former is provisioned, and can include, for example, firewall rules or intrusion detection rules. The latter, called *flow state* is created on-the-fly when packets of a new flow are received for the first time or through an explicit request. Flow state can vary in size. For example, on seeing a packet from a new flow, a middlebox may generate some small state like a firewall pinhole or a NAT translation entry, or it may begin to maintain a buffer to reconstruct a TCP stream.

Flow state is stored in a *flow table* data structure and accessed using flow identifiers (packet headers) as keys. (Figure 1) Models of middleboxes have been developed that represent state as a key-value database indexed by

addresses (e.g., a standard IP 5-tuple) [12]. A middlebox may have multiple flow tables (e.g., per network interface). It may also contain timers that refer to flow state, for example, to clean up stale flows.

We have performed a detailed analysis of the source code or specifications of several middleboxes to confirm that they fit into this model. We discuss three of them below:

Bro. Bro [19] is a highly stateful intrusion detection system. It maintains a flow table in the form of a dictionary of `Connection` objects, indexed by the standard IP 5-tuple without the protocol field. Inside the `Connection` objects, flow-related state varies depending on the protocol analyzers that are being used. Analyzer objects contain state machine data for a given protocol (e.g., HTTP) and reassembly buffers to reconstruct a request/response payload, leading to tens of kilobytes per flow in the common case. A dictionary of timers is maintained for each `Connection` object. Bro also contains statistics and configuration settings.

Application Delivery Controller (ADC). ADC [35, 38, 40] is a packet-modifying load balancer that ensures the addresses of servers behind it are not visible to clients. It contains a flow table that is indexed by the source IP address and port. Flow-specific data includes the internal address of the target server and a timestamp, resulting in only tens of bytes per flow. ADC also maintains timers for each flow, which it uses to clean up flow entries.

Stateful NAT64. Stateful NAT64 [15] translates IPv6 packets to IPv4 and vice-versa. NAT64 maintains three flow tables, which it calls session tables: for UDP, TCP, and ICMP Query sessions, respectively. Session tables are indexed using a 5-tuple. Flow state, called session table entries (STEs), consists of a source and destination IPv6 address and a source and destination IPv4 address, so is therefore tens of bytes in size. Timers, called STE lifetimes, are also maintained.

2.2 The Split/Merge Abstraction

The Split/Merge abstraction enables transparent and balanced elasticity for virtual middlebox applications. Using Split/Merge, middlebox applications can continue to be written and configured oblivious to the number of replicas that may be instantiated. Each replica perceives an identical VM abstraction, down to the details of the MAC address on the virtual network interface card.

As depicted Figure 2, using Split/Merge, the output of a middlebox application remains *consistent*, regardless of the number of replicas that have been instantiated or destroyed throughout its operation. Slightly more formally:

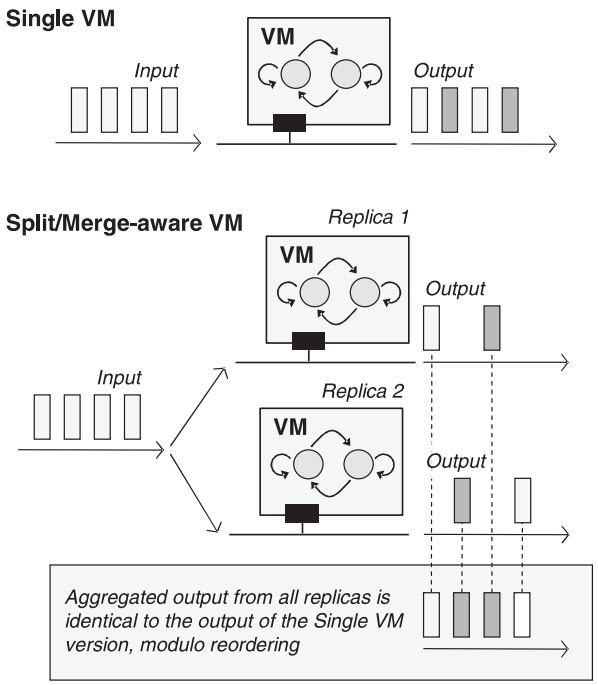


Figure 2: Split/Merge retains output consistency irrespective of the number of replicas.

Definition. *Let a VM be represented by a state machine that accepts input from the network, reads or writes some internal state, and produces output back to the network. A Split/Merge-aware VM is abstractly defined as a set of identical state machine replicas; the aggregate output of which—modulo some reordering—is identical to that of a single machine, despite the partitioning of the input between the replicas. Consistency is achieved by ensuring that each replicated state machine can access the state required to produce the appropriate output in response to its share of the input.*

There are two types of state in a Split/Merge-aware VM (Figure 1): *internal* and *external* state. Internal state is relevant only to a single replica. It can also be thought of as "ephemeral" [5]; its contents can deviate between replicas of the state machine without affecting the consistency of the output. Examples of internal state include background operating system processes, cache contents, and temporary side effects. External state, on the other hand, transcends a single replica. If accessed by *any* replica, external state cannot deviate from what it would have been in a single, non-replicated state machine without affecting output consistency. For example, a NAT may look up the port translation for a particular flow. Any deviation in the value of this state would cause the middlebox to malfunction, violating consistency.

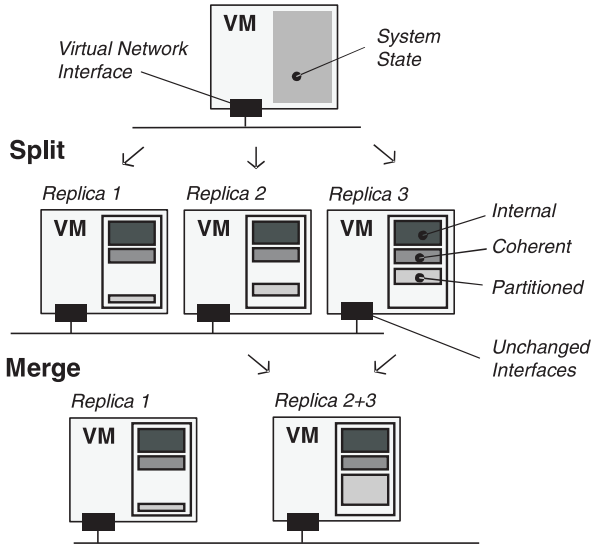


Figure 3: A Split/Merge-aware VM uses the different types of state to achieve transparent elasticity.

As depicted in Figure 1, external state can take two forms: *partitioned* or *coherent*. Partitioned state is made up of a collection of sub-states, each of which are intrinsically tied to a subset of the input and therefore only need to be accessed by the state machine replica that is handling that input. The NAT port translation state is an example of partitioned state, because only the replica handling the network flow in question must access the state. Coherent state, on the other hand, is accessed by multiple state machine replicas, regardless of how the input is partitioned. In Figure 1, the flow table and timers reside in partitioned state, while configuration information and statistics reside in coherent state.

2.3 Using Split/Merge for Elasticity

Figure 3 depicts how the state of a middlebox VM is split and merged when elastically scaling out and in. On scale-out, internal state is replicated with the VM, but begins to diverge as each replica runs independently. Coherent state is also replicated with the VM, but remains consistent (or eventually consistent) because access to coherent state from each replica is transparently coordinated and controlled. Partitioned state is split among the VM replicas, allowing each replica to work in parallel with its own sub-state. At the same time, the input to the VM is partitioned, such that each replica receives only the input pertinent to its partitioned sub-state.

On scale-in, one of the replicas is selected to be destroyed. Internal state residing at the replica can be safely discarded, since it is not needed for consistent output. Coherent state may be discarded when any outstanding updates are pushed to other replicas. The sub-states of the partitioned state residing at the dying replica are merged into a surviving replica. At the same time, the input that was destined for the dying replica is also redirected to the surviving replica now containing the partitioned sub-state.

2.4 Challenges

To implement a system that supports Split/Merge for virtual middleboxes, several challenges need to be met.

C1. **VM state must be classified.** For virtual middlebox applications to take advantage of Split/Merge, each application must identify which parts of its VM state are internal vs. external. Fortunately, the structure of middleboxes (Figure 1) is naturally well-suited to this task. The flow table of middleboxes already associates partitioned state with a subset of the input, namely network flows.

C2. **Transactional boundaries must be respected.** In some cases, a middlebox application may need to convey that it finished processing relevant input before partitioned state can be moved from one VM to another. For example, an IDS may continuously record information about a connection's state; such write operations must complete before the state can be moved. Other cases, such as a NAT looking up a port translation, do not have such transactional constraints.

C3. **Partitioned state must be able to move between replicas.** Merging partitioned state from multiple replicas requires at the most primitive level the ability to move the responsibility for a flow from one replica to another. In addition to moving the flow state, the replica receiving the flow must update its flow table data structures and timer structures so that it can readily access the state.

C4. **Traffic must be routed to the correct replica.** As partitioned state—associated with network flows—is split between VM replicas, the network must ensure that the appropriate flows arrive at the replica holding the state associated with those flows. Routing is complicated by the fact that partitioned state may move between replicas and each replica shares the same IP and MAC address.

The Split/Merge abstraction can be thought of in two parts: splitting and merging *VM state* between replicas (Figure 3), and splitting and merging *network input* between replicas (Figure 2). As such, the challenges can also be classified into those that deal with state management (C1, C2, C3) and those that deal with network management (C4).

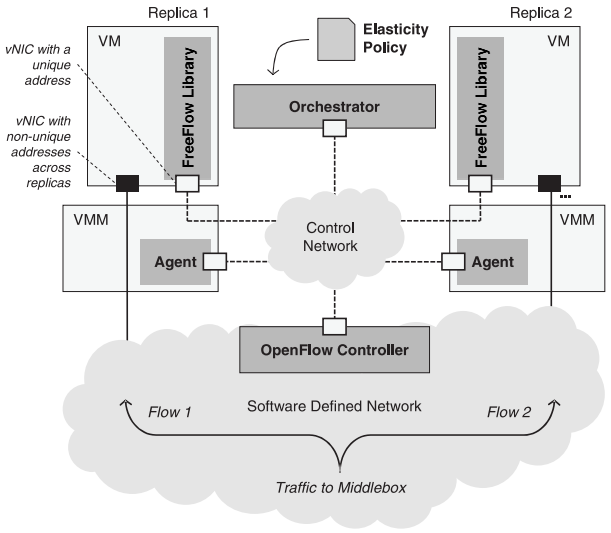


Figure 4: FreeFlow Architecture

```
// MIDDLEBOX−SPECIFIC PARTITIONED STATE HANDLING

create_flow(flow_key, size);     // alloc flow state
delete_flow(flow_key);           // free flow state

flow_state get_flow(flow_key); // increment refcnt
put_flow(flow_key);              // decrement refcnt

flow_timer(flow_key, timeout, callback);


// COHERENT STATE HANDLING

create_shared(key, size, cb); // if cb is null, then use
delete_shared(key);           // strong consistency

state get_shared(key, flags); // synch | pull | local
put_shared(key, flags);       // synch | push | local
```

Figure 5: Interface to the FreeFlow library

3 FreeFlow

FreeFlow implements the Split/Merge abstraction to enable virtual middleboxes to achieve transparent, balanced elasticity. The design of FreeFlow is shown in Figure 4. It consists of four components. First, the state aspects of the Split/Merge abstraction are implemented via the application-level *FreeFlow library*, which addresses the state-related challenges (C1, C2,C3). In particular, through the interface to the library, a middlebox application classifies its state as internal or external and communicates its transactional requirements. Additionally, the library manages all aspects of external state, including the migration of partitioned sub-states. Second, the network aspects of the Split/Merge abstraction are implemented in FreeFlow's *Split/Merge-aware software defined network* (SDN). The SDN addresses the final challenge (C4) and ensures that the correct network flows are routed to the replica maintaining the corresponding partitioned sub-state. Third, the *orchestrator* implements an elasticity policy: it decides when to create or destroy VM replicas and when to migrate flows between them. Finally, *VMM agents* perform the actual creation and destruction of replicas. The four components communicate with each other over a control network, distinct from the Split/Merge-aware SDN.

We have implemented a prototype of FreeFlow, including all of its components shown in Figure 4. Each physical machine runs Xen [2] as a VMM and Open vSwitch [20] as an OpenFlow-compatible software switch. In all components, flows are identified using the IP 5-tuple.

3.1 Guest Library: State Management

Middlebox applications interact with the FreeFlow library in order to classify state as external and identify transaction boundaries on such state. The interface to the library is shown in Figure 5. Behind the scenes, the library interfaces with the rest of the FreeFlow system to split and merge partitioned state between replicas and control access to coherent state.

To fulfill the task of identifying external state, the library acts a memory allocator, and is therefore the only mechanism the middlebox application can use to obtain partitioned or coherent sub-state. Partitioned state in middlebox applications generally consists of a flow table and a list of timers related to flow state; therefore, the library manages both. The library provides an interface, `create_flow` to allocate a new entry in the flow table against a *flow key*, which is usually an IP 5-tuple. A new timer (and its callback) can be allocated against a flow key using `flow_timer`. Coherent sub-state is allocated against a key by invoking `create_shared`, but the key is not necessarily associated with a network flow.

Transaction boundaries are inferred by maintaining reference counts for external sub-states. Using `get_flow` or `get_shared`, the middlebox application accesses external sub-state from the library, at which point a reference counter (refcnt) is incremented. When the application finishes with a transaction on the sub-state, it informs the library with `put_flow` or `put_shared`, which decrements the reference counter. The application must avoid dangling references to partitioned state. If it fails to inform the library that a transaction is complete, the state will be pinned to the current replica.

The library may copy partitioned sub-state across the

control network to another replica in response to a notification from the orchestrator (§ 3.3). When instructed to migrate a flow—identified with a flow key and a unique network address for a target replica on the control network—the library waits for the reference counter on the state to become zero, then copies the flow table entry and any timers across the control network. The flow table at the source is updated to record the fact that the particular flow state has migrated. Upon future `get_flow` calls, the library returns an error code indicating that the flow has migrated and the packet should be dropped. Similarly, when the target library receives flow data—and the flow key for it to be associated with—during a flow migration, the flow table and timer list are updated and the orchestrator is notified. At any one time, only one library instance maintains an active copy of the flow data for a particular flow.

The library also manages the consistency of coherent state across replicas. In most cases, strong consistency is not required. For example, the application can read and write counters or statistics locally most of the time (using the `LOCAL` flag on `get_shared`). Periodically, the application may require a consistent view of a counter. For example, an IDS may need to check an attack threshold value has not been exceeded. For periodic merging of coherent state between replicas, FreeFlow supports *combiners* [7, 16]. On `create_shared`, an application can specify a callback (`cb`) function, which takes a list of coherent state elements as an argument and combines them in an application specific way. In most cases, this function simply adds the values of the counters in the coherent state. The combiner will be invoked automatically by the library when a replica is about to be destroyed. It can also be invoked explicitly by the application either before a reference to the coherent state is obtained (using the `PULL` flag on `get_shared`) or after a transaction is complete (using the `PUSH` flag on `put_shared`). The combiner never runs in the middle of a transaction; `get_shared` using `PULL` may block until other replicas finish their transaction and the state can be safely read. In the rare case that strong consistency is required, the application does not specify a combiner, and library instead interacts with a distributed locking service [3,11]. On `get_shared` (with the `SYNCH` flag), the library obtains the lock associated with the specified key and ensures that it has the most recent copy of the coherent data. The library releases the lock on `put_shared` and the system registers the local copy of the coherent data as the most recent version.

We have implemented the FreeFlow library as a C library. In doing so, we addressed the implementation challenge of allowing flow state to include self-referential pointers to other parts of the flow state. To support unmodified pointers, the library must ensure that the flow state resides at same virtual address range regardless of which replica it is in. To accomplish this, the library allocates a large virtual address space *before* notifying the VMM agent to compute the initial snapshot. Within the virtual address range, the orchestrator provides each replica with a non-overlapping region to service new flow related memory allocations obtained with `create_flow`.

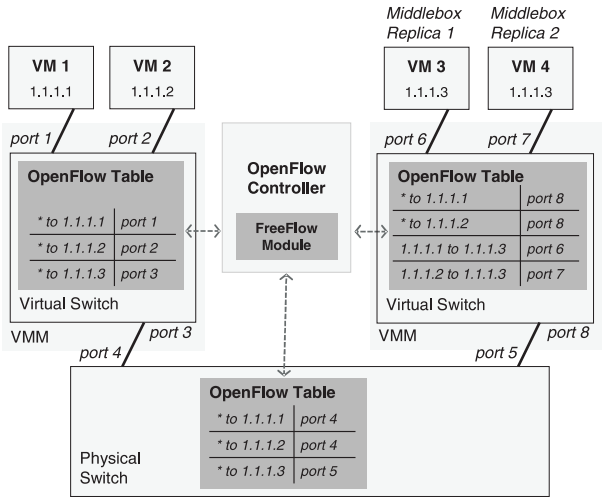


Figure 6: The SDN splits network input to replica VMs based on flow rules. The SDN ensures that traffic from VM 1 arrives at VM 3 and traffic from VM 2 arrives at VM 4. For clarity, we have omitted the flow rules for routing middlebox output.

3.2 Split/Merge-Aware SDN: Network Management

The Split/Merge-aware SDN implements the networking part of the Split/Merge abstraction. Each replica VM contains an identical virtual network interface. In particular, every replica has the same MAC and IP address. Maintaining consistent addresses in the replicas avoids breaking OS or application level address dependencies in the internal state within a VM.

As depicted in Figure 6, FreeFlow leverages OpenFlow-enabled [41] network elements (e.g., switches [20], routers) to enforce routing to various replicas. As packets flow through the OpenFlow network, each network element searches a local forwarding table for rules that match the headers of the packet, indicating they belong to a particular flow. If an entry is found, the network element forwards the packets along the appropriate interface on the fast path. If no entry exists, the packet (or just its header) is forwarded to an *OpenFlow controller*. The OpenFlow controller has a global view of the network and can make a routing decision for the new flow. The controller then pushes a new rule to one or more network

elements so that future packets belonging to the flow can be forwarded without consulting the controller.

The Split/Merge-aware SDN must ensure that packets arrive at the appropriate replica even as partitioned flow state migrates between replicas. To do this, FreeFlow contains a customized OpenFlow controller that communicates with the orchestrator (§ 3.3). When a flow is migrated between replicas, the orchestrator interfaces with the OpenFlow controller to communicate the new forwarding rules for the flow. Packets belonging to new flows are forwarded to the OpenFlow controller by default. The OpenFlow controller picks a replica toward which the new flow should be routed and notifies the orchestrator.

When a flow migration notification is received from the orchestrator, rules to route the flow are deleted from all network elements in the current path traversed by the flow. The flow is then considered *suspended*. Packets arriving from the switches are temporarily buffered at the OpenFlow controller until the flow is *resumed* by the controller, at the new replica. The flow is not resumed until partitioned sub-state has arrived at its new destination. The controller resumes a flow by calculating a new path for the flow that traverses the new replica, installing forwarding rules in the switches on the path, and injecting any buffered packets directly into the virtual switch connected to the new replica.[1]

We implemented the SDN in a module on top of POX [42], a python version of the popular NOX [10] OpenFlow controller. The controller provides a simple web API that allows it to receive notifications from the orchestrator about events like middlebox creation and deletion, or instructions to migrate one or more flows from one replica to another. We addressed three implementation challenges. First, the controller cannot use MAC learning techniques for middleboxes because every replica shares a MAC address. Instead, when replicas are created, the VMM agent registers a replica interface on a virtual switch port with the controller. Second, ARP broadcast requests may cause multiple replicas to respond or unexpected behavior, since they share a MAC address. To avoid this, the controller intercepts and replies to ARP requests that refer to the middlebox IP. Finally, the controller decides which replica a new flow is routed to, so must ensure that bi-directional flows are assigned to the same replica. This is achieved by maintaining a table that maps each flow to its replica that is checked before assigning new flows to replicas.

[1] Alternately, buffering could occur at the destination hypervisor and the controller could update the path immediately upon suspend, thereby reducing its load.

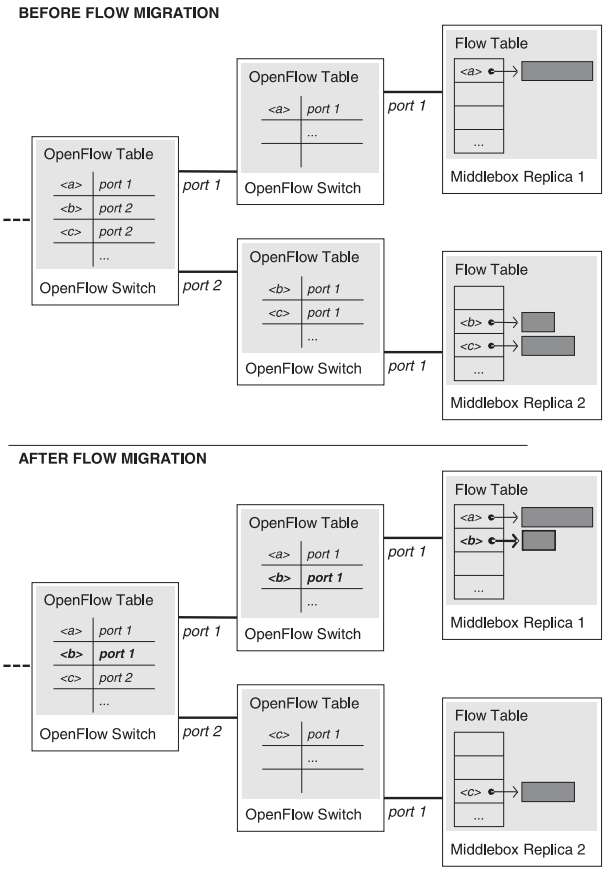


Figure 7: Migrating flow $\langle b \rangle$ from Replica 2 to Replica 1

3.3 Orchestrator: Splitting and Merging

The orchestrator implements the most fundamental primitive for enabling the splitting and merging of partitioned state between replicas: flow migration. Figure 7 shows the migration of a flow $\langle b \rangle$ between two replicas. The orchestrator interacts with other parts of the system as follows. It:

- instructs the SDN to suspend the flow $\langle b \rangle$ such that no traffic of the flow will reach either replica.
- instructs the guest library in Replica 2 to transfer the partitioned state associated with $\langle b \rangle$ to Replica 1.
- instructs the SDN to resume the flow by modifying the routing of flow $\langle b \rangle$ such that any new traffic belonging to the flow will arrive at Replica 1.

It is possible, although rare in practice, that some packets will arrive at Replica 2 after the flow state has been migrated to Replica 1. For example, packets may be buffered in the networking stack in the kernel of Replica 2 and not yet have reached the application. In case the application receives a packet after the flow state

is migrated, it should drop the packet.[2]

The orchestrator triggers the creation or destruction of replicas by the VMM Agent (§3.4) in order to scale in or out as part of an elasticity policy. It can trigger these operations automatically based on utilization (resembling the Autoscale functionality in Amazon EC2 [34]) or explicitly in response to user input.

3.4 VMM Agent: Scaling In and Out

The VMM agent creates or destroys replica VMs in response to instructions from the orchestrator. Replica VMs are instantiated from a point-in-time snapshot of the first instantiation of the middlebox VM, before it began processing packets. During library initialization, after variables in the internal state are initialized but before the VM has allocated any external state, the FreeFlow library instructs the VMM agent to compute the snapshot. By definition, the internal state in the replica VM can diverge from the snapshot.

We have implemented the VMM agent in Xen's control domain (Domain 0). Communication with the library is implemented using Xenstore, a standard, non-network based virtual communication channel used between guest VMs and Domain 0 in Xen. Checkpoints are computed with the `xm save` command, and replicas are instantiated with the `xm restore` command.[3] The time to create a new fully operational replica is on the order of a few seconds; it may be possible to reduce this delay with rapid VM cloning techniques [14].

3.5 Limitations

Virtual middleboxes cannot use FreeFlow (or the Split/Merge paradigm in general) unless their structure roughly matches that described in Figure 1. While most middleboxes we have examined do fit this architecture, it should be noted that some middleboxes are more difficult to adapt to FreeFlow than others. The main cause of difficulty is how the middleboxes deal with partitioning granularity and coherent state.

Middleboxes can be composed of numerous layers and modules, each of which may refer to flows using a different granularity. For example, in an IDS, like Bro, one module may store coarse-grained state (e.g., concerning all traffic in an IP subnet), while another may store fine-grained state (e.g., individual connection state). There are two approaches to adapting such a middlebox to FreeFlow. First, the notion of a flow could be expanded to the largest granularity of all modules. In the preceding example, this would mean using the same flow key for all data related to all flows in an IP subnet, fundamentally limiting FreeFlow's ability to balance load. Second, a fine-grained flow key could be used to identify partitioned state, causing the coarse-grained state to be classified as coherent. If strong consistency is required for the coarse-grained state or a combiner cannot be specified, this approach may cause high overhead due to state synchronization.

[2] In this case, the library returns an error code when flow-specific state is accessed (§ 3.1).

[3] In our prototype, the distribution of VM disk images to physical hosts is performed manually.

4 Experience Building Split/Merge Capable Middleboxes

To validate that the Split/Merge abstraction is well suited to virtual middleboxes, we have ported Bro, an open-source intrusion detection system, to run on FreeFlow. To evaluate a wider range of middleboxes, we have also implemented two synthetic FreeFlow middleboxes.

4.1 Bro

Bro is composed of two key components: an Event Engine and a Policy Script Interpreter. Packets captured from the network are processed by the Event Engine. The Event Engine runs a protocol analysis, then generates one or more predefined events (e.g., connection establishment, HTTP request) as input to the Policy Script Interpreter. The Policy Script Interpreter executes code written in the Bro scripting language to handle events. As explained in Section 2.1, the Event Engine maintains a flow table with each table entry corresponding to an individual connection. Each event handler executed by the Policy Script Interpreter also maintains state that is related to one or more flows.

Our porting effort focused on Bro's Event Engine and one event handler.[4] The event handler scans for potential SQL injection strings in HTTP requests to a webserver. The handler tracks—on a per-flow basis—the number of HTTP requests (`num_sqli`) that contain a SQL injection exploit. When `num_sqli` exceeds a predefined threshold (`sqli_thresh`), Bro issues an alert.

Porting Bro to FreeFlow. Porting Bro to FreeFlow involved the straightforward classification of external state and interfacing with the FreeFlow library to manage it. First, we identified all points of memory allocation in the code. If the memory allocation was for flow-specific data, we modified the allocation to use FreeFlow-provided memory instead of the heap. In certain cases, we had to provide custom implementations of standard C++ constructs like `std::List`, to avoid leaking references to FreeFlow-managed memory.

[4] For ease of implementation, we ported the event handler to C++ instead of using the Bro scripting language.

After ensuring partitioned state was allocated in FreeFlow-managed memory, we checked for external references to it. The only two references were from the global dictionary of `Connection` objects and the global dictionary of timers. Since FreeFlow manages access to flow-related objects and timers, we could replace these two global collections. We found that Bro always accesses flow-related state in the context of processing a single packet, and therefore has well-defined transactional boundaries. References from FreeFlow-managed classes to external memory occur only to read static configuration data (internal state).

As expected, there was very little data that we classified as coherent state. We used FreeFlow's support for combiners for non-critical global statistics counters. The combiners were configured to only be invoked by the system (i.e., on replica VM destruction). We did not find any variables that required strong consistency or real-time synchronization across replicas.

Verification. To validate the correctness of the modified system, we used a setup consisting of a client and a webserver, separated by two middlebox replicas running the modified version of Bro. At a high level, we used the client to issue a single flow of HTTP requests containing SQL injection exploits while FreeFlow migrated the flow between the two replicas multiple times. We check for the integrity of state and execution by ensuring (a) Bro generates an alert, (b) the number of exploits detected exactly matches those sent by the client (c) both replicas remain operational after each flow migration. Assuming Bro sees all packets on the flow, the first two conditions cannot be satisfied if the state becomes corrupted during migration. Additionally, the system would crash on flow migration when objects inside FreeFlow memory refer to external memory that does not exist on the local replica.

4.2 Synthetic Middlebox Applications

We built two synthetic FreeFlow based middlebox applications that capture the essence of commonly used real world middlebox applications. The first application is compute-bound. It performs a set of computations on each packet of a flow, resembling the compute intensive behavior of middlebox applications like an Intrusion Prevention System (IPS) or WAN optimizer. The second application modifies packets in a flow in both directions, using a particular application-level (layer 7) protocol, resembling a NAT or Application Layer Gateway. Both middleboxes were built in userspace using the Linux netfilter [39] framework to interpose on packets arriving at the VM. The userspace applications inspect and/or modify packets before forwarding them to the target.

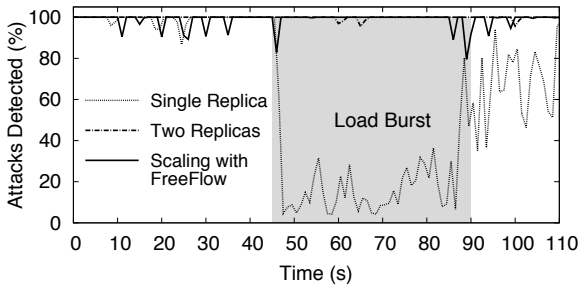


Figure 8: Splitting/Merging Bro for Stateful Elasticity

5 Evaluation

FreeFlow enables balanced elasticity by leveraging the Split/Merge abstraction to distribute—and migrate—flows between replicas. In this section, we evaluate FreeFlow with the following goals:

- demonstrate FreeFlow's ability to provide dynamic and stateful elasticity to complex real world middleboxes (§ 5.1),
- demonstrate FreeFlow's ability to alleviate hotspots created by a highly skewed load distribution across replicas (§ 5.2),
- measure the gain in resource utilization when scaling in a deployment using FreeFlow (§ 5.3), and
- quantify the performance overhead of migrating a single flow under different application loads (§ 5.4).

In our experimental setup, a set of client and server VMs are placed on different subnets. Traffic—TCP or UDP—is routed between the VMs via a middlebox. We evaluate FreeFlow using Bro or one of the synthetic middleboxes described in Section 4.2.

5.1 Stateful Elasticity with Split/Merge

Figure 8 shows FreeFlow's ability to dynamically scale Bro out and in during a load burst, splitting and merging partitioned state. In this experiment, the generated load contains SQL injection exploits; we measure the percentage of attacks detected by Bro to determine Bro's ability to scale to handle the load burst.

Load is generated by a configurable number of cURL-based [36] HTTP clients in the form of a continuous sequence of POST requests to a webserver. The requests contain SQL injection exploits; an attack comprises 31 consecutive requests. Each client is configured to generate 50 requests/second. Throughout the experiment (for 120 seconds), 30 clients generate a base load. We inject a load burst 45 seconds into the experiment by introducing an additional 30 clients and 10 UDP flows

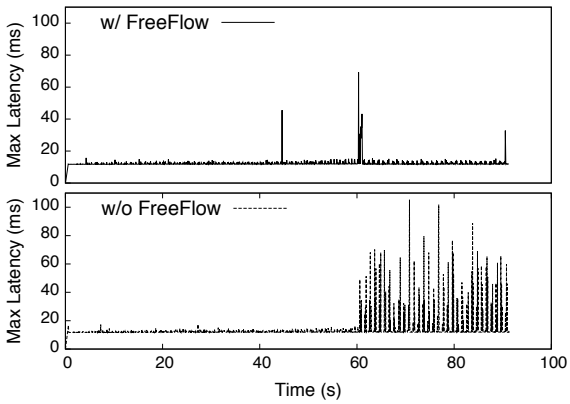


Figure 9: Eliminating hotspots with FreeFlow

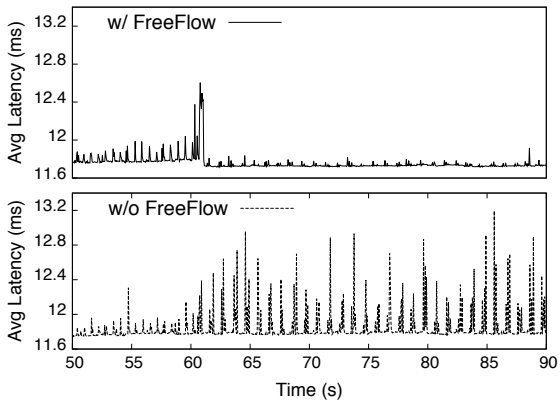


Figure 10: Performance impact of FreeFlow's load rebalancing on hotspots

(1 Mbps each) that do not contain attacks. The load burst lasts 45 seconds, after which the additional client and UDP traffic ceases.

We compare three scenarios: a single Bro instance that handles the entire load burst, a pair of Bro replicas that share load (flows are assigned to replicas in a round-robin fashion), and Bro running with FreeFlow. The FreeFlow scenario begins with a single replica and FreeFlow is configured to create a new replica and split flows and state between them when the number of flows handled by the replica exceeds 60. Similarly, it is configured to merge flows and state and destroy a replica when the number of flows handled by a replica drops below 40.

As shown in Figure 8, until the load burst at $t = 45$s, all three configurations have a 100% detection rate. During the load burst, the performance of the single replica reduces drastically because packets are dropped and attacks are missed. The two replica cluster does not experience any degradation as it has enough capacity and the load is well balanced between the two replicas.

The FreeFlow version of Bro behaves in the same manner as a single replica, until the load burst is detected around $t = 45$s. While partitioned state is being split to a new replica, packets are dropped and attacks are missed. However, the detection rate quickly rises because the two replicas have enough capacity for the load burst. After the load burst ($t = 85$s), FreeFlow detects a drop in load, so merges partitioned state and destroys one of the replicas. The FreeFlow version of Bro continues to detect attacks at the base load with a single replica. FreeFlow therefore enables Bro to handle the load burst without wasting resources by running two replicas throughout the entire experiment.

5.2 Hotspot Elimination

In this experiment, we demonstrate FreeFlow's ability to eliminate hotspots that arise when the load distribution across middleboxes becomes skewed. For the purpose of this discussion, we define a hotspot as the degradation in network performance due to high CPU or network bandwidth utilization at the middlebox.

We use the compute-bound middlebox application described in Section 4.2 under load from 1 Mbps UDP flows. We define our scale-out policy to create a new replica once the number of flows in a replica reaches 100 (totaling 100 Mbps per replica). Flows are gradually added to the system every 500 ms up to a total of 101 flows. After scaling out, the system has two replicas: one with 100 flows and another with just one flow.

As expected, the replica handling 100 flows experiences much higher load than the other replica. The resulting hotspot is reflected by highly erratic packet latencies experienced by the clients, shown in Figure 9 and Figure 10. Figure 9 shows the maximum latency, while Figure 10 shows the fluctuations in the average latency during the last 40s of the experiment. FreeFlow splits the flows evenly among the two replicas thereby redistributing the load and alleviating the hotspot. Ultimately, FreeFlow achieves a 26% reduction in the average maximum latency during the hotspot, with a 73% lower standard deviation.

Irrespective of flow duration and traffic patterns, without FreeFlow's ability to balance flows, an over-conservative scale-out policy may be used to ensure hotspots do not occur, leading to low utilization and wasted resources. By balancing flows, FreeFlow enables less conservative scale-out policies leading to higher overall utilization.

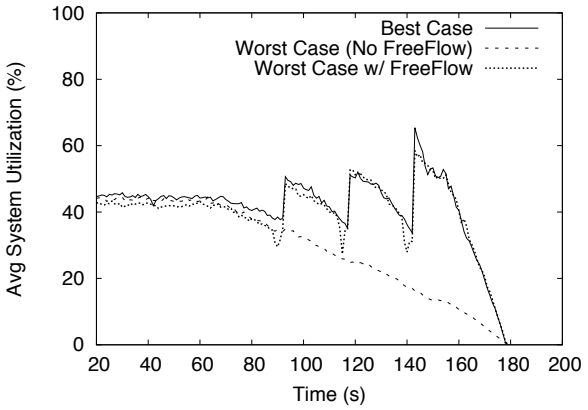


Figure 11: Scaling in with FreeFlow

5.3 Efficient Consolidation

In this experiment, we show how FreeFlow's ability to statefully merge flows between two or more replicas can be used to consolidate resources during low load and improve overall system utilization. We measure how quickly FreeFlow can scale in compared to a standard *kill-based* technique, in which a replica is killed only when all its flows have expired. We also measure the average system utilization per live replica during scale in, shown in Figure 11.

We start with 4 replicas running the compute-bound middlebox application (§4.2), handling 50 UDP flows of 1 Mbps each. One flow expires every 500 ms according to a best case or worst case scenario.

In the best case scenario, the first 50 flows expire from the first replica in the first 25 seconds, enabling the kill-based technique to destroy the replica. The second 50 flows expire from the second replica in the next 25 seconds, enabling the second replica to be destroyed, and so on. In this case, the average system utilization remains high throughout the scale-in process, with a sawtooth pattern as shown in Figure 11.

In the worst case scenario, flows expire from replicas in a round-robin fashion. In a kill-based system, each of the 4 replicas contains one or more flows until the very end of the experiment, preventing the system from destroying replicas. This results in steadily degrading average system utilization over the duration of the experiment.

On the other hand, even in the worst case, FreeFlow can destroy a replica every 25 seconds. To accomplish this, FreeFlow is configured with a scale-in policy that triggers once the average number of flows per replica falls below 50. When scaling in, FreeFlow kills a replica after merging its state and flows with the remaining replicas. Subsequently, in the worst case, FreeFlow maintains average system utilization close to that of the kill-based strategy in the best case scenario and improves the average system utilization by up to 43% in the worst case scenario. Based on the the time at which the first replica was killed in the worst case scenario, FreeFlow can scale in 50% faster than the standard kill-based system.

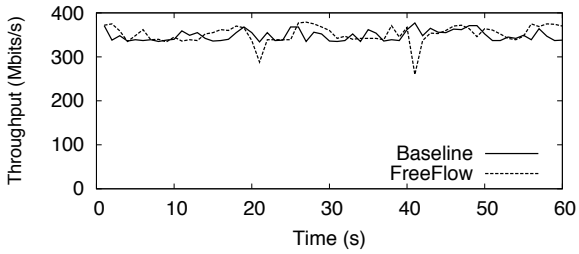


Figure 12: Impact of flow migration on TCP throughput (migration at 20s & 40s)

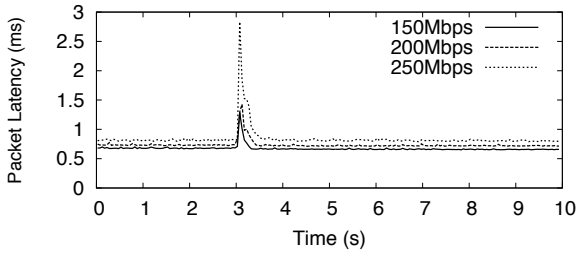


Figure 13: Latency overhead of flow migration

FreeFlow does impact the performance of flows during the experiment; in particular, packet drops are caused by flow migrations that happen when a replica is merged. However, performance impact is low: the average packet drop rate per-flow was 0.9%.

5.4 Migrating Application Flow State

Flow state migration is a fundamental unit of operation in FreeFlow, when splitting or merging partitioned state between replicas. Figure 12 shows the impact on TCP throughput during flow migration compared to a baseline where no migration is performed. We use the Iperf [37] benchmark to drive traffic on a single TCP stream between the client and the server, through the compute-bound middlebox. We perform two flow migrations: one at 20th and another at 40th second, respectively. When sampled at 1 second intervals, we observe a 14 – 31% drop in throughput during the migration period, lasting for a maximum of 1 second.[5]

We further study the overhead of flow migration on a

[5] Due to Iperf's limitation on the minimum reporting interval, (1 second), we are unable to calculate the exact duration of the performance impact.

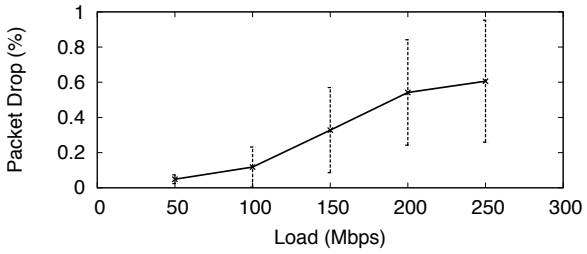


Figure 14: Packet drop rate during flow migration

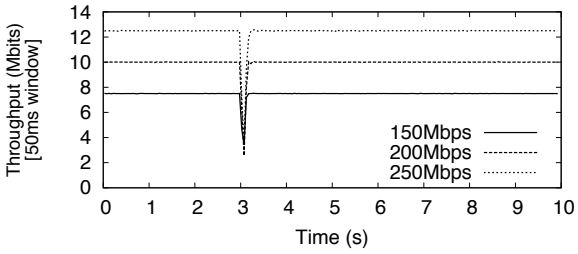


Figure 15: Throughput overhead of flow migration (50 ms window)

single UDP flow using the packet modifier middlebox application (§4.2). For these experiments, the flows are 10 seconds in duration and the migration is initiated after three seconds from the start of the flow. The impact of a single flow migration on end-to-end latency for different flow rates is shown in Figure 13. We observe a maximum of 1 ms increase in latency during flow migration. The latency fluctuations last for a very short period of time (500 ms). Figure 14 shows the overall packet drop rate for the entire duration of the flow. The overall packet drop rate is less than 1% including any disruption caused by the migration. Figure 15 shows the impact on throughput as observed by the client, when the flow migration occurs. The plotted throughput is based on a 50 ms moving window. As the load on the network increases, there is an increase in throughput loss due to flow migration. However, the drop in throughput occurs only for a brief period of time and quickly ramps up to pre-migration levels.

6 Related Work

Split/Merge relies on the ability to identify per-flow state in middleboxes. The behavior and structure of middleboxes has been characterized through the use of models [12]. In other work, state in middleboxes has been identified as global, flow-specific, or ephemeral (per-packet) [30]. On a single machine granularity, MLP [31], HILTI [26], and multi-threaded Snort [21, 22] all exploit the fact that flow-related processing rarely needs access to data for other flows or synchronization with them. CoMb [23] exploits middlebox structure to consolidate heterogeneous middlebox applications onto commodity hardware, but does not address the issue of scaling, parallelism, or elasticity.

Clustering techniques have traditionally been used to scale-out middleboxes. The NIDS Cluster [28] is a clustered version of Bro [19] that is capable of performing coordinated analysis of traffic, at large scale. By exposing policy layer state and events as serializable state [27], individual nodes are able to obtain a global view of the system state. The NIDS Cluster cannot scale dynamically and statefully, as it lacks the ability to migrate lower layer (event engine) flow state and their associated network flows across replicas.

FreeFlow leverages OpenFlow in its Split/Merge-aware SDN. Load balancing has been implemented in an SDN using OpenFlow with FlowScale [25], and wildcard rules can accomplish load balancing in the network while reducing load on the controller [32]. The Flowstream architecture [8] includes modules—for example, VMs—that handle flows and can be migrated, relying on OpenFlow to redirect network traffic appropriately. However, Flowstream does not characterize external state within an application. Olteanu and Raiciu [18] similarly attempt to migrate per-flow state between VM replicas without application modifications.

There are many ways in which different types of applications are dynamically scaled in the cloud [29]. Knauth and Fetzer [13] describe scaling up general applications using live VM migration [6] and oversubscription. Amazon's Autoscaling [34] automatically creates or destroys VMs when user-defined thresholds are exceeded. SnowFlock [14] provides sub-second scale-out using a *VM fork* abstraction. These approaches do not enable balancing of existing load between instances, potentially resulting in load imbalance [33].

7 Conclusion

We have described a new abstraction, Split/Merge, and a system, FreeFlow, that enables transparent, balanced elasticity for stateful virtual middleboxes. Using FreeFlow, middleboxes identify partitioned state, which can be split among replicas or merged together into a single replica. At the same time, FreeFlow partitions the network to ensure packets are routed to the appropriate replica. As networks become increasingly virtualized, FreeFlow addresses a need for elasticity in middleboxes, without introducing the configuration complexity of running a cluster of independent middleboxes. Further, as virtual servers become increasingly mobile, utilizing live VM migration across or even between data centers, the

ability to migrate flows—or split and merge them between replicas—will become even more important.

8 Acknowledgments

We would like to thank our shepherd, Mike Freedman, and the anonymous referees for their helpful comments.

References

[1] ANANTHANARAYANAN, G., KANDULA, S., GREENBERG, A., STOICA, I., LU, Y., SAHA, B., AND HARRIS, E. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proc. of USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (2010).

[2] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the Art of Virtualization. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (2003).

[3] BURROWS, M. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proc. of USENIX Symposium on Operating Systems Design & Implementation (OSDI)* (2006).

[4] CARPENTER, B., AND BRIM, S. Middleboxes: Taxonomy and Issues. RFC 3234, `https://tools.ietf.org/rfc/rfc3234.txt`, 2002.

[5] CHANDRA, R., ZELDOVICH, N., SAPUNTZAKIS, C., AND LAM, M. S. The Collective: A Cache-Based System Management Architecture. In *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)* (2005).

[6] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live Migration of Virtual Machines. In *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)* (2005).

[7] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *Communications of ACM 51*, 1 (2008).

[8] GREENHALGH, A., HUICI, F., HOERDT, M., PAPADIMITRIOU, P., HANDLEY, M., AND MATHY, L. Flow Processing and the Rise of Commodity Network Hardware. *ACM SIGCOMM Computer Communications Review 39*, 2.

[9] GU, Y., SHORE, M., AND SIVAKUMAR, S. A Framework and Problem Statement for Flow-associated Middlebox State Migration. `http://tools.ietf.org/html/draft-gu-statemigration-framework-02`, 2012.

[10] GUDE, N., KOPONEN, T., PETTIT, J., PFAFF, B., CASADO, M., MCKEOWN, N., AND SHENKER, S. NOX: Towards an Operating System for Networks. *ACM SIGCOMM Computer Communications Review 38*, 3.

[11] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proc. of USENIX Annual Technical Conference (ATC)* (2010).

[12] JOSEPH, D. A., AND STOICA, I. Modeling Middleboxes. *IEEE Network 22*, 5 (2008).

[13] KNAUTH, T., AND FETZER, C. Scaling non-elastic Applications Using Virtual Machines. In *IEEE International Conference on Cloud Computing* (2011).

[14] LAGAR-CAVILLA, H. A., WHITNEY, J. A., SCANNELL, A. M., PATCHIN, P., RUMBLE, S. M., DE LARA, E., BRUDNO, M., AND SATYANARAYANAN, M. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proc. of ACM European Conference on Computer Systems (EuroSys)* (2009).

[15] M. BAGNULO, P. MATTHEWS, I. V. B. Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers. `http://tools.ietf.org/id/draft-ietf-behave-v6v4-xlate-stateful-12.txt`, 2010.

[16] 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 *Proc. of ACM SIGMOD* (2010).

[17] MELL, P., AND GRANCE, T. The NIST Definition of Cloud Computing. In *National Institute of Standards and Technology Special Publication 800-145* (2011).

[18] OLTEANU, V. A., AND RAICIU, C. Efficiently Migrating Stateful Middleboxes. In *ACM SIGCOMM - Demo* (2012).

[19] PAXSON, V. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks 31*, 23-24 (1999).

[20] PFAFF, B., PETTIT, J., KOPONEN, T., AMIDON, K., CASADO, M., AND SHENKER, S. Extending Networking into the Virtualization Layer. In *Proc. of ACM Workshop on Hot Topics in Networks* (2009).

[21] ROESCH, M. Snort - Lightweight Intrusion Detection for Networks. In *Proc. of USENIX Conference on System Administration* (1999).

[22] SCHUFF, D. L., CHOE, Y. R., AND PAI, V. S. Conservative vs. optimistic parallelization of stateful network intrusion detection. In *Proc. of ACM Symposium on Principles and Practice of Parallel Programming* (2007).

[23] SEKAR, V., EGI, N., RATNASAMY, S., REITER, M. K., AND SHI, G. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. of USENIX Symposium on Networked Systems Design & Implementation (NSDI)* (2012).

[24] SHERRY, J., AND RATNASAMY, S. A Survey of Enterprise Middlebox Deployments. Tech. Rep. UCB/EECS-2012-24, EECS Department, University of California, Berkeley, 2012.

[25] SMALL, C. FlowScale. GENI Engineering Conference (Poster), `http://groups.geni.net/geni/attachment/wiki/OFIU-GEC12-status/FlowScale_poster.pdf`, 2012.

[26] SOMMER, R., CARLI, L. D., KOTHARI, N., VALLENTIN, M., AND PAXSON, V. HILTI: An Abstract Execution Environment for Concurrent, Stateful Network Traffic Analysis. Tech. Rep. TR-12-003, ICSI, 2012.

[27] SOMMER, R., AND PAXSON, V. Exploiting Independent State For Network Intrusion Detection. In *Proc. of Computer Security Applications Conference* (2005).

[28] VALLENTIN, M., SOMMER, R., LEE, J., LERES, C., PAXSON, V., AND TIERNEY, B. The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In *Proc. of International Conference on Recent Advances in Intrusion Detection* (2007).

[29] VAQUERO, L. M., RODERO-MERINO, L., AND BUYYA, R. Dynamically Scaling Applications in the Cloud. *ACM SIGCOMM Computer Communications Review 41*, 1.

[30] VERDÚ, J., NEMIROVSKY, M., GARCÍA, J., AND VALERO, M. Workload Characterization of Stateful Networking Applications. In *Proc. of International Symposium on High-Performance Computing* (2008).

[31] VERDÚ, J., NEMIROVSKY, M., AND VALERO, M. MultiLayer Processing - An Execution Model for Parallel Stateful Packet Processing. In *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2008).

[32] WANG, R., BUTNARIU, D., AND REXFORD, J. OpenFlow-based Server Load Balancing Gone Wild. In *Proc. of USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services* (2011).

[33] WELSH, M., AND CULLER, D. Adaptive Overload Control for Busy Internet Servers. In *Proc. of USENIX Symposium on Internet Technologies and Systems* (2003).

[34] Amazon EC2: Auto Scaling. `http://aws.amazon.com/autoscaling/`.

[35] BIG-IP Local Traffic Manager (LTM). `http://www.f5.com/products/big-ip/big-ip-local-traffic-manager/`.

[36] libcurl - The Multiprotocol File Transfer Library. `http://www.tcpdump.org/`.

[37] Iperf: TCP and UDP Bandwidth Performance Measurement Tool. `http://iperf.sourceforge.net/`.

[38] Linux Virtual Server. `http://www.linuxvirtualserver.org/`.

[39] Netfilter Packet Filtering Framework. `http://www.netfilter.org`.

[40] Citrix NetScaler ADC. `http://www.citrix.com/netscaler`.

[41] The OpenFlow Switch Specification. `http://www.openflow.org`.

[42] POX OpenFlow Controller. `http://www.noxrepo.org/pox/about-pox/`.

PinPoint: Localizing Interfering Radios

Kiran Joshi, Steven Hong, Sachin Katti
{*krjoshi, hsiying, skatti*}*@stanford.edu*

Abstract

This paper presents PinPoint, a technique for localizing rogue interfering radios that adhere to standard protocols in the inhospitable ISM band without any cooperation from the interfering radio. PinPoint is designed to be incrementally deployed on top of existing 802.11 WLAN infrastructure, and used by network administrators to identify and troubleshoot sources of interference which may be disrupting the network. PinPoint's key contribution is a novel algorithm that accurately computes the line of sight angle of arrival (AoA) and cyclic signal strength indicator (CSSI) of the target interfering signal at all APs, even when the line of sight (LoS) component is buried by stronger multipath components, interference and noise. PinPoint leverages this algorithm to design an optimization technique, which can localize interfering radios and simultaneously identify the type of interference. Unlike several localization techniques which require extensive pre-deployment calibration (e.g. RF-Fingerprinting), PinPoint requires very little calibration by the network administrator, and uses a novel algorithm to self-initialize its bearings, even if the locations of some AP are initially unknown and are oriented randomly. We implement PinPoint on WARP software radios and deploy in an indoor testbed spanning an entire floor of our department. We compare PinPoint with the best known prior RSSI [8, 11] and MUSIC-AoA based approaches and show that PinPoint achieves a median localization error of 0.97 meters, which is around three times lower compared to the RSSI [8, 11] and MUSIC-AoA based approaches.

1 Introduction

Interference is the number one cause for poor wireless performance. All of us have had anecdotal experiences, where, even though the AP is quite close, we experience poor performance and more often that not, interference is to blame. Yet, in spite of these pervasive problems, we often know very little about where this interference is coming from. We do not know the nature of the interfering radio (e.g. whether it is another WiFi network, Bluetooth or Zigbee), neither do we know where it is located. Without such localization, troubleshooting performance problems becomes hard.

One might imagine that we could leverage the extensive prior work [4, 8, 11, 15, 20, 21, 22, 23, 24, 25] that has tackled indoor localization. However, none of it is applicable to localizing interfering radios. First, most of them are RSSI based and work typically with WiFi, i.e. they measure the RSSI of the WiFi signal from multiple vantage points, and then leverage propagation models and triangulation techniques to localize. However, when localizing interference, the source could often be a non-WiFi radio. Further, its unlikely we can get a good estimate of the interfering signals RSSI because there could be multiple signals present from concurrent transmitting radios. Another class of RSSI techniques requires extensive RF fingerprinting of the indoor environment. However these techniques do not work under interference either since the RSSI fingerprints will be distorted when there are multiple concurrent transmissions. Further, these techniques are expensive to deploy since they require constant and recurring site fingerprinting. Another class of techniques [16, 17, 18, 19, 24] use non-RSSI based techniques such as range-finding and time of arrival, however all of them require modifications to and cooperation from the client radio (e.g in the form of special beaconing hardware), which is untenable when we are trying to localize an interferer not under our control.

In this paper we present PinPoint, a system that computes the nature as well as the location of the interfering radio(s) with *sub-meter accuracy*. PinPoint is robust, it can localize each interfering radio even when multiple interfering radios may be transmitting concurrently. Furthermore, PinPoint's accuracy is at least two times better as compared to RSSI based techniques *even when no interference is present*. Hence even though PinPoint's design is motivated by the scenario of localizing interference, it provides a general indoor localization technique that works across a wide variety of scenarios. The system consists of an indoor AP infrastructure with PinPoint capability, of which a small subset ($3-5$ per floor of a large department building) are anchor APs that already know their absolute indoor location. The PinPoint APs work together to detect and localize interfering radios. PinPoint assumes no co-operation from the interfering radio, works with legacy client radios, and it does not assume any knowledge of the protocol, power or the spectrum at which interfering radios are transmitting. Further, PinPoint does not require any expensive calibration or surveying, either at installation or in subsequent operation. We believe this combination of accuracy, robustness and generality is a first.

PinPoint's key contribution is a novel algorithm that ac-

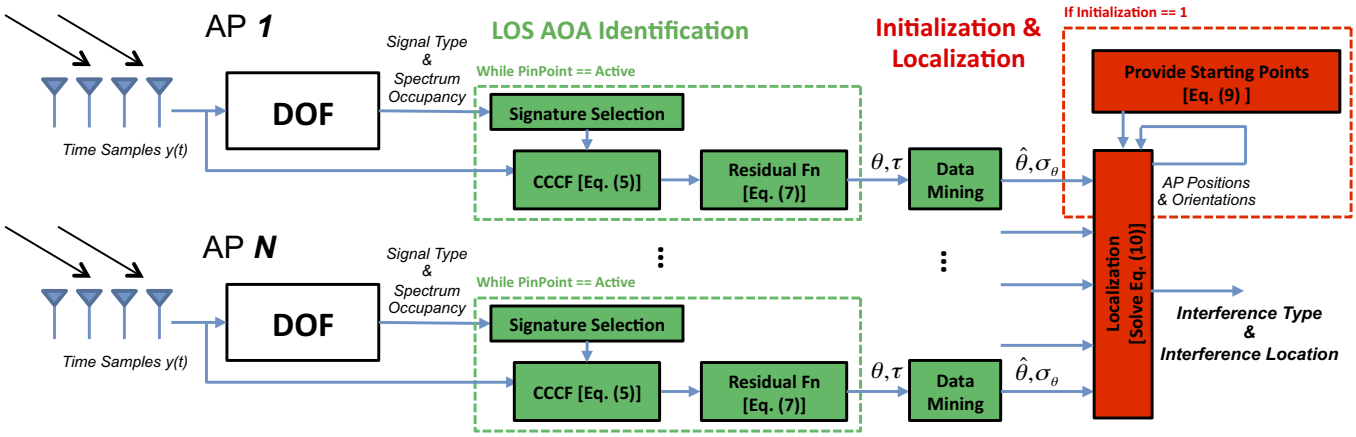


Figure 1: **PinPoint Architecture:** PinPoint is deployed incrementally on top of existing WLAN infrastructure. PinPoint first leverages DOF [2] to identify and separate out multiple sources of interference. PinPoint then uses this information to build a novel algorithm which detects the LOS AoA, even when it is buried by stronger multipath reflections. Finally, PinPoint leverages the LOS AoA at individual APs by aggregating all of the measurements at a central server and solving an optimization algorithm which triangulates the source of interference.

curately computes the LoS AoA and the signal strength (CSSI) of only the target interfering radio signal at all APs where the LoS component is at least barely perceptible (i.e. its signal strength is at least $-10dB$ relative to the strongest path). Once the LoS AoA and the corresponding CSSI is estimated at a few of the APs, PinPoint runs an optimization algorithm based on triangulation to compute the exact location. Computing the LoS AoA and CSSI in practice, however, is challenging because of two inherent environmental factors. First, since we have no control over the interfering radios, the APs could be receiving signals that consist of contributions from multiple interfering radios, potentially using different physical layer protocols. Second, radio signals bounce off walls and other objects, and create numerous multi-path components that arrive at the AP at a variety of angles. Often, the strongest component of the received signal will be a reflection and the LoS component might be weak due to obstructions. Hence, PinPoint's algorithm has to disentangle the LoS AoA and CSSI in spite of these factors that make the signal look like it is coming at the AP from a variety of sources and a variety of angles.

Our key insight is that both multipath and angles of arrival manifest themselves as relative delays between copies of the same signal arriving at an AP from the target radio. For example, since the LoS component will have the shortest path to the AP, it will arrive before any reflected component. Similarly, a signal arriving at a particular AoA at an AP, will arrive at slightly different times at the different antennas in a multiple antenna AP because the signal has to travel slightly different distances. We design novel algorithms based on cyclostationary signal analysis [2, 1] that can exploit these relative delays. Specifically we isolate the LoS component by finding the relative delay between the first time we see a signal and when its reflection arrives. Next, we find the relative delays at which the isolated LoS component arrives at different antennas at the AP, and from that infer the AoA of the LoS component. The cyclostationary signal analysis also allows us to accurately infer the signal strength of the target interfering radio without much contribution from the noise and signal sources that do not bear the same cyclic signature as the interfering radio, we will refer to this as CSSI in the rest of the paper.

PinPoint's main conceptual contributions are novel algorithms to accurately, efficiently and robustly extract these relative delays and LoS AoAs and CSSI measurements from noisy interfered signals. As far as we are aware, no prior localization technique has been able to isolate and compute accurate LoS estimates in the presence of severe multipath and interference. We implement PinPoint using standard WARP software radios [6] equipped with 4 antennas as the RF hardware. We evaluate PinPoint using testbed experiments in an indoor environment with typical multipath and interference and compare it against the state of the art RSSI based approach [8, 11] and MUSIC-AoA algorithm. We find that:

- PinPoint is significantly more accurate than both the RSSI and MUSIC-AoA approaches. In our testbed experiments PinPoint's median error is 0.97 meters, while the RSSI and MUSIC-AoA approaches achieve median errors of 3.35 meters and 2.94 meters respectively.
- PinPoint is even more accurate when there is no secondary interference and a single target radio is being localized, it achieves a median error of 0.83 meters, while the RSSI and MUSIC-AoA approaches achieve 2.32 meters and 2.9 meters respectively. Thus even though PinPoint's original design goal was to localize interference, it provides a general and

accurate technique for all localization problems.

- PinPoint works accurately even if the AP deployment is sparse. In our testbed, by default we used five APs to cover an entire floor (this was the number recommended by our network manager to provide WiFi coverage for the floor). However, we found that even if we used only three APs, PinPoint still achieves a median error of 1.76 meters, thus providing good accuracy even in sparse deployments.

Finally, while PinPoint has significant advantages over RSSI based approaches, it does require that the APs perform extra DSP computation to calculate LoS AoAs and CSSIs. While this does not require any extra RF hardware (such as filters, synchronization circuitry etc), it does require extra compute horsepower at the APs. RSSI based approaches do not, they can directly use the RSSI estimate from the AP. We believe this cost is modest, in fact, Cisco has started adding similar interference detection (but not localization) capability to its enterprise APs, and given the unique features PinPoint provides, the extra cost is quite reasonable.

2 PinPoint: Overview

Fig. 1 shows the overall PinPoint architecture. PinPoint assumes that it is deployed on multiple antenna APs (4 antennas are sufficient in our current prototype). Further, we assume that a small number of the APs (3-5) act as anchor nodes and know their absolute location. Both these assumptions are easily satisfied, almost all new AP deployments use MIMO APs with at least four antennas [7], and finding the location at the time of deployment for a few APs (e.g. the ones near a window where GPS works) is relatively straightforward.

To localize interfering radios, PinPoint has to deal with two major challenges

- Most likely, we are not going to have a priori knowledge about the interfering radios. We cannot assume we know their transmit power, the frequencies or even the protocol they are using (e.g. WiFi, Bluetooth, Zigbee etc). Further, we cannot ask these radios to send special beacon packets for localization when we need them to. Consequently existing RSSI based techniques are difficult to apply, under interference even measuring the RSSI of an individual radio's signal is hard.
- In indoor environments, where PinPoint is likely to be employed, a localization system has to deal with multipath effects and the lack of strong LoS paths between the radios and the APs. Specifically, an interfering radio may not have a visual LoS path to any AP (e.g. the AP is outside your office). Further, in the ISM band radio signals will bounce off walls and other objects and arrive at the AP from multiple directions.

PinPoint deals with both these challenges and is more accurate than any existing localization system under these scenarios. At a high level, PinPoint's localization algorithm proceeds with the following steps:

1. **Identify the source of interference:** PinPoint takes the received signal and first identifies the nature of the interfering radio (e.g. whether it is WiFi, Bluetooth or Zigbee). To do so, PinPoint builds upon prior work (DOF) in interference identification [2] to discriminate between the signals of different interference types.
2. **Compute the Line of Sight Angle of Arrival (LoS AoA) For Each Interfering Source:** PinPoint next computes all the AoAs at which the interfering radio's signal is arriving at an AP. PinPoint uses a novel technique to compute the AoA of only the LoS component of a radio signal, even when the LoS path is obstructed. PinPoint does not compute AoA corresponding to the non-LoS paths, which are not useful for localization, thus reducing computation power compared to methods like SAGE [5].
3. **Compute the Cyclic Signal Strength Indicator (CSSI) for each interfering Source:** PinPoint also computes the signal strength of only the interfering sources that have been identified in step 1.
4. **Localize the interfering radio:** PinPoint then collects the LoS AoA and CSSI measurements from multiple APs in the deployment, and runs a triangulation based optimization algorithm to compute the location of the interfering radio. Note that this requires that we know the location of the APs themselves in advance, however requiring that the operator measures the absolute location of all the APs during deployment is cumbersome. Instead, PinPoint leverages the above techniques to *localize the APs* themselves at the time of deployment. PinPoint only requires that we know the location of a few (typically 3-5 suffice) anchor APs at the time of deployment. Such computed AP locations are then used in the localization of interfering radios.

For step 1, PinPoint builds on prior work (DOF) in interference identification based on cyclostationary signal analysis [2], while this paper designs novel algorithms for the other three steps. In the next section, we describe how the first three steps above are performed, followed by a discussion of the localization algorithms in Sections 4 and 5.

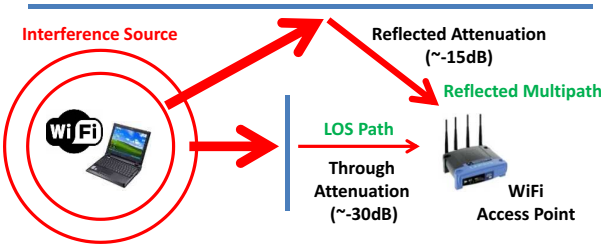


Figure 2: **Obstructed LOS and Multipath:** The LOS path, even when it is obstructed, is the first to impinge on the AP. But when there are reflected paths which are stronger (NLOS scenario), they can mask the LOS component reflections. PinPoint applies novel techniques to detect this LOS component, even when it is >10dB weaker than the reflected paths.

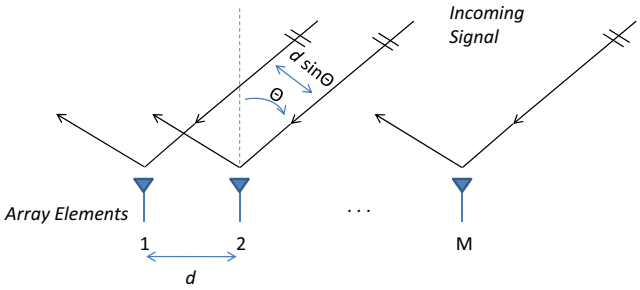


Figure 3: **Uniform Linear Array:** The delays/phase shifts experienced at each antenna is proportional to the AoA.

3 Design: Computing LoS AoA and CSSI

PinPoint's design is based on the insight that multipath effects and different AoAs manifest themselves in relative delays at which the signal arrives at the AP. For example, a multipath environment shown in Fig. 2 results in signal reflections, and thus the AP receives multiple copies of the signal at different delays depending on the relative delay between the different paths. However, the LoS component (even if it passes through an obstruction) will have the shortest path to the AP and hence will arrive first, assuming the obstruction does not completely block the signal. It may be weak however, relative to some unobstructed multipath reflection. Thus there will be a relative delay between the LoS component (which might be relatively quite weak) and the first multipath reflection component.

Similarly, different angles of arrival manifest themselves as relative delays at which the same signal arrives at different antennas. Fig. 3 demonstrates the idea for a linear multiple antenna array. Since the antenna #1 is a bit further away than the antenna #2 for the given AoA, the signal hitting the antenna #2 will take a little bit longer to hit the antenna #1 and so on. Thus if one knew the relative delay that a signal component took between impinging on two consecutive antennas, we can infer the AoA of that signal component.

Based on the above geometrical insights, we invent a novel algorithm for identifying the angle of arrival of the line of sight component of a signal. The algorithm proceeds in two steps

- First, it isolates the component of the signal that corresponds to the LoS path by leveraging the insight that this will be very likely the first component to arrive at the AP.
- Next, it repeats the above step at each antenna at the AP, and then correlates the isolated LoS components across all the antennas with each other to infer their relative delay, and thus the AoA corresponding to that component. By construction, this will correspond to the AoA of the LoS component.

Our first goal therefore is to isolate the LoS component of the signal by exploiting the insight that it will be the first to arrive at the AP. However, there are several challenges to accomplish this step. First, the AP does not know the interfering signal, as it does not know what data is encoded, what rate it is sent at, what modulation is used and so on. Hence relying on such properties to identify the LoS component is not possible. Second, the LoS component may be too weak due to obstructions, or interfered by other signals.

To tackle this challenge, PinPoint exploits signatures that result from hidden repeating patterns in the signal obtained by cyclostationary signal analysis. The signatures are robust as they can be detected even when the signal is very weak [1], or even when it is interfered with [1]. As discussed before, PinPoint builds on DOF [2], an interference identification system based on cyclostationary signal analysis.

PinPoint then designs novel algorithms that leverage these cyclostationary signatures to *determine the LoS AoA and CSSI, even in severely obstructed environments dominated by multipath components or under heavy interference*. PinPoint exploits the knowledge of the signal types to correlate known signatures with the received signals. Note that this does not imply that we know the interfering signal's contents, only that we know how the underlying structure of the signal has patterns independent of the information that the signal is carrying.

In the following section, we'll describe the above process in detail. We will begin by providing a quick primer on how the hidden repeating patterns within wireless signals can be leveraged to form unique signatures for every signal type [2]. We will then explore how these signatures can be exploited to determine the LOS AoA and CSSI.

3.1 Multipath Signal Model

We start with a more formal description of how both multipath and different angles of arrival manifest themselves as relative delays between copies of the same signal arriving at the AP. This description is well known, but serves to set up the context in which PinPoint operates.

The scattering caused by the indoor environment causes each signal to traverse multiple paths to our APs. The multipath arrivals are shifted and scaled copies of the same signal, occurring at varying angles and delays. We can explicitly model the total signal impinging on a single antenna at our AP as

$$y_1(t) = \sum_{k=1}^{L} s_k(t) + n_1(t) \quad (1)$$

where L is the number of multipath components, $n_1(t)$ represents additive noise at the antenna, and

$$s_k(t) = \alpha_k s(t - d_k) e^{2\pi i f_c (t - d_k)} \quad (2)$$

represents each multipath component, where $s(t)$ is the transmitted signal, $\alpha_k \in \mathbf{R}$ is the attenuation for each arriving path, d_k is the time delay of each path, and f_c is the carrier frequency used in the transmission.

Assuming each AP has multiple antenna, the delays in propagation paths between each of the antennas enables us to measure the varying angles of each multipath component. The delay is a function of the antenna arrangement and for exposition simplicity, we consider a Uniform Linear Array (ULA), which is an array that has all of its antennas on a line with equal half-wavelength ($\frac{\lambda}{2}$) spacing between the antennas. In the ULA configuration, the signal arriving at the i^{th} antenna has a difference in propagation path that results in a time delay of $(i-1)\frac{\lambda \sin\theta}{2c}$, where c is the speed of propagation through the medium. The output of the antenna array in response to the L multipath signals can be expressed as,

$$\begin{aligned}
y_1(t) &= \textstyle\sum_{k=1}^{L} s_k(t) + n_1(t) \\
y_2(t) &= \textstyle\sum_{k=1}^{L} s_k(t) e^{j2\pi f_c \frac{\lambda \sin\theta_k}{2c}} + n_2(t) \\
&\vdots \\
y_M(t) &= \textstyle\sum_{k=1}^{L} s_k(t) e^{j2\pi f_c (M-1) \frac{\lambda \sin\theta_k}{2c}} + n_M(t)
\end{aligned}$$

This can be written is vector form as,

$$y(t) = \sum_{k=1}^{L} s_k(t) a(\theta_k) + n(t), \quad (3)$$

where $y(t) \in \mathbf{C}^M$ is the received vector, $n(t) \in \mathbf{C}^M$ is the noise vector, and $a(\theta)$ is the steering vector of the array given by

$$a(\theta) = \left[e^0 \; e^{j2\pi f_c \frac{\lambda \sin\theta_k}{2c}} \; \ldots \; e^{j2\pi f_c (M-1) \frac{\lambda \sin\theta_k}{2c}} \right]^T.$$

3.2 Leveraging Knowledge of Signal Type

PinPoint builds on DOF [2], an interference identification system that leverages the hidden and repeating patterns that are unique and necessary for operation and are present in all wireless protocols. DOF builds on prior work in cyclostationary signal analysis [1] and leverages the following idea from that work: *if a signal has a repeating pattern, then if we correlate the received signal against itself delayed by a fixed amount, the correlation will peak when the delay is equal to the period at which the pattern repeats*. Specifically, let's denote the raw signal samples we are receiving by $x[n]$. Consider the following function

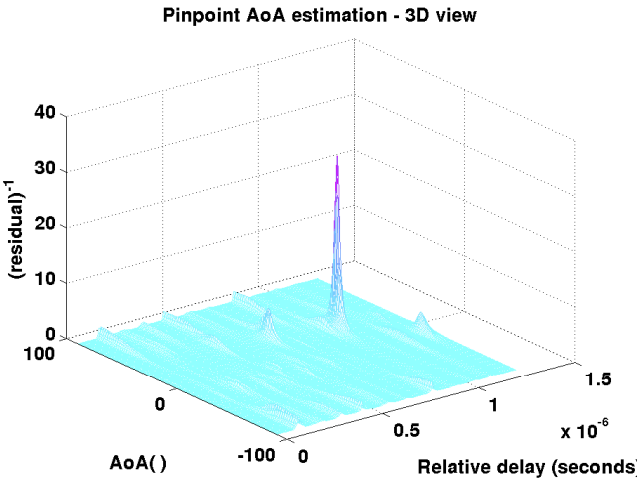


Figure 4: **Binning in Time/AoA:** The CCCF embeds both the delay and AoA of every arriving propagation path - as shown above the residual function (eq. 7) peaks at the angles and relative delays of each path. By searching for the first peak with the minimum delay, we can detect the LOS component's AoA.

$$R_x^{\alpha}(\tau) = \sum_{n=-\infty}^{\infty} x(n) x^*(n-\tau) e^{-j2\pi\alpha n} \quad (4)$$

For an appropriate value of τ corresponding to the time period between the repeating patterns, the above value will be maximized, since the repeating patterns in $x[n]$ will be aligned. Further, these peak values occur only at periodic intervals in n. Hence the second exponential term $e^{-j2\pi\alpha n}$ is in effect computing the frequency α at which this hidden pattern repeats. We define such a frequency as a *pattern frequency*, and (4) is known as the Cyclic Autocorrelation Function (CAF) [1] at a particular pattern frequency α and delay τ. The CAF will exhibit a high value only for delays and pattern frequencies that correspond to repeating patterns in the signal.

Because each wireless protocol utilizes a different set of parameters (encoding, modulation, etc.), each protocol exhibits a unique set of repeating patterns and therefore have unique signature CAFs. Hence, DOF uses machine learning heuristics to uniquely identify different signal types. We omit the details of how DOF accomplishes this for brevity and refer the reader to [2] for a more detailed description. For our purposes it suffices to know that PinPoint uses DOF to identify the signal type.

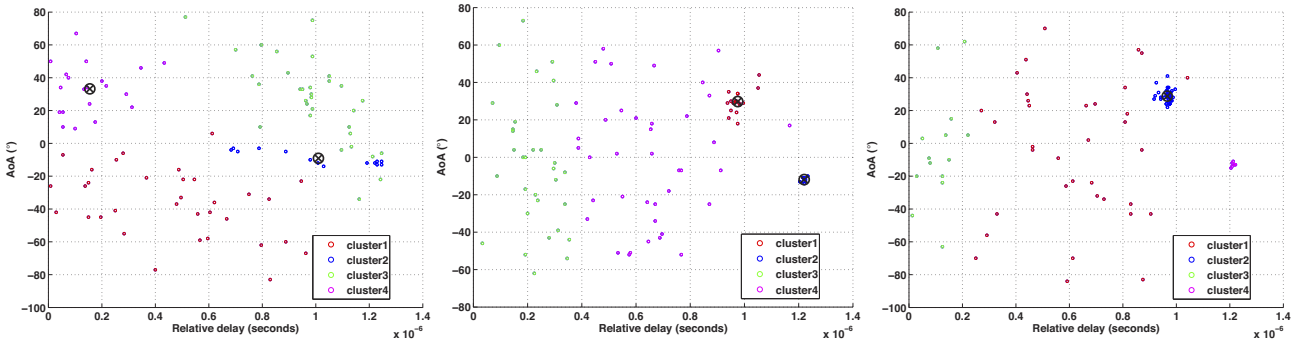


Figure 5: **Data clustering/mining for LOS AoA when the received power differences between the LOS path and the strongest multipath is (A) -10dB (B) 0dB (C) 10dB:** The direct LOS AoA is arriving at 35°, while the strongest multipath component is arriving at −10°. You can see that even when the direct LOS component is more than 10dB weaker than the strongest multipath component in **(A)**, PinPoint is still able to detect the LOS AoA. When the LOS AoA is stronger, it is of course easier to detect and PinPoint does well in these scenarios as expected.

3.3 Line of Sight AoA Identification

The challenge is to identify the LOS component ($s_1(t)$) and its corresponding AoA (θ_1) even when it is significantly weaker than the multipath components. To do this, PinPoint leverages what is already known - specifically the type of interfering source and thus the pattern frequencies at which the signatures repeat, enabling us to create a test signature. PinPoint has a set of test signatures corresponding to the expected set of interfering radios (e.g. one signature for 802.11, another for ZigBee, etc.) which exhibit features at the corresponding pattern frequencies for each protocol. Note that these signatures do not assume that the data in the target interfering signal is known, they are merely creating a dummy signal which has the same repeating patterns as the identified signal type. Further, there is a different test signature for each pattern frequency. In other words since WiFi exhibits hidden repeating patterns at several pattern frequencies, there is a separate test signature for each pattern frequency in WiFi.

Once the type has been identified by DOF, it is cross correlated against the corresponding signature for a particular pattern frequency. Specifically, we can calculate the cross correlation between our target signal $y_i(t)$ and our test signature $s_T(t)$ using the following function [10]:

$$R^{\alpha}_{y_i s_T}(\tau) = \sum_{t=-\infty}^{\infty} y_i(t) s^*_T(t-\tau) e^{-i2\pi\alpha t} \tag{5}$$

Unlike the CAF, the Cyclic Cross Correlation Function (CCCF) peaks at values of τ corresponding to the relative delays between the multipath components. The reason is because the multipath signal is a linear combination of copies of the same signal shifted in time due to reflections. When the test signature is aligned with one of the multipath components, in effect the hidden repeating patterns in the signature and the received signal align and the CCCF peaks. Thus the first peak in the CCCF will be for the signal component that is received first, i.e. likely the LoS component, the next peak is for the first reflected component and so on. The relative distance between the peaks thus corresponds to the relative delays between multipath components. The benefit of using the CCCF is that it provides robust detectable peaks even when the received signal is very weak or interfered with, because the hidden repeating patterns allow us to integrate and eliminate the uncorrelated noise and interference to produce a robust peak.

When we apply the CCCF to the all signals of the antenna array, we obtain a function which is dependent on the pattern characteristics (τ), the delay between the multipath components (d_k), and the angles at which each path impinges (θ_k)

$$R^{\alpha}_{y_i s_T}(\tau) = \sum_{k=1}^{L} \beta_k R^{\alpha}_{s_T s_T}(\tau - d_k) a(\theta_k), \tag{6}$$

$$\text{where} \qquad \beta_k = \alpha_k e^{-\pi i \alpha d_k} e^{-2\pi i f_c d_k}.$$

We leverage this fact to form a residual function which is a function of both the delay and the angle of arrival:

$$res^{\alpha}_k(\tau,\theta) = \sum_{m=1}^{M} \left| \frac{R^{\alpha}_{y_m s_T}(\tau)}{R^{\alpha}_{y_k s_T}(\tau)} - \frac{a_m(\theta)}{a_k(\theta)} \right|^2, \; k = \{1,...,M\}. \tag{7}$$

Observe that in (6), the delays (τ) at which the function typically peaks at are shifted by the physical propagation delay experienced by each multipath component d_k. Thus when $\tau = d_k$, the first term in the residual function, $\frac{R^{\alpha}_{y_m s_T}(\tau)}{R^{\alpha}_{y_k s_T}(\tau)}$, will become the ratio of the steering vectors, as β_k and $R^{\alpha}_{s_T s_T}(\tau)$ are canceled because the patterns in the signal are identical to the ones in the signature. The second term then cancels with the first when the value of θ matches the value of each multipath's AoA.

We can leverage this insight to form an optimization problem that computes the LoS AoA. Specifically, if we solve

$$\hat{\theta}_1 = \underset{\{(\tau,\theta):\tau_{min}\leq\tau\leq\tau_{max},\theta_{min}\leq\theta\leq\theta_{max}\}}{\text{argmax}} \frac{1}{\sum_\alpha \sum_{k=1}^{M} res_k^\alpha(\tau,\theta)} \quad (8)$$

where $[\tau_{min},\tau_{max}]$ and $[\theta_{min},\theta_{max}]$ are the range of interest for the unknown variables τ and θ respectively. The output of the above optimization is an estimate of the AoA of the LoS component, with the relative delay between the LoS component and the first multipath component. Fig. 4 shows the result of this optimization.

3.3.1 Mining Multiple Measurements Across Time

The above optimization provides a noisy estimate of LoS AoA and its relative delay. In order to minimize the uncertainty, PinPoint performs the optimization (8) separately over multiple packets received from the same source. By running PinPoint over time for different sequences of data, we can build sets of relative delay and AOA pairs. We found empirically that these sets can be clustered to find an accurate estimate of the LoS AoA, if it exists and is perceptible (i.e. if it has a signal strength of at least -10 dB). However, if the LoS component is extremely weak (less than -10dB signal strength) perhaps because of a strong obstruction, we found that the computed relative delays and AoAs do not cluster and are all over the place. PinPoint leverages this insight to eliminate signals where a perceptible LoS component does not exist.

Algorithmically, we use a clustering technique based on Gaussian mixture models. Results of this clustering for various scenarios are shown in Fig. 5. After clustering, PinPoint checks if there are multiple clusters, and then calculates the mean and standard deviation for each cluster. Prioritizing the minimization of false positives, we discarded clusters which did not possess a minimum number of data points and clusters with AoA standard deviation above a certain threshold.These steps are not necessary for the operation of PinPoint but helps to fine-tune the AoA estimates. Of the remaining clusters, the mean AoA corresponding to the cluster which has the smallest relative delay is declared to be θ_1, the AoA of the direct LOS component.

3.4 Computing the Cyclic Signal Strength Indicator (CSSI)

PinPoint leverages cyclostationary analysis to compute the signal strength of only the target interfering radios. This is different from traditional RSSI, those techniques will not work in our context because in the presence of interference those techniques cannot measure the RSSI of the different constituent signals making up the interference. PinPoint on the other hand can leverage its ability to isolate the target interfering signal using cyclostationary signal analysis (the CAF and the CCCF functions), and then use the correlation values themselves as a proxy for the relative strength of that signal arriving at different APs. Note that the stronger the target signal , the higher the correlation value. Hence instead of trying to measure the agggregate signal strength, we can simply use the correlation values at different APs to represent the strength contributed by only the target interfering radio. We call this correlation value cyclic signal strength indicator (CSSI).

Plugging the relative delay of the LOS component τ into eq. (5) and taking the magnitude of $R^\alpha_{y_i s_T}(\tau)$ gives us a value that is a proxy of the signal strength of the target radio, which PinPoint can use to further constrain its localization search as we'll show in the next section. Note that for localization we do not need to know the actual RSSI as long as the value we use as a proxy exhibits the same attenuation pattern as RSSI. PinPoint's localization only needs to compare the relative RSSI across multiple APs, and for that the proxy computed above suffices.

4 Initializing PinPoint

PinPoint collects the LoS AoA and RSSI measurements from multiple APs in the enterprise deployment, and runs a triangulation based optimization algorithm to compute the location of the interfering radio.

The challenge is that the above process implicitly assumes that we know the location of the APs themselves. However, enterprise WiFi networks often consist of tens to potentially several hundreds of APs. Providing the precise location of each AP is cumbersome since GPS signals are unreliable indoors and orientation is similarly tricky since most APs are not equipped with compasses. At best, the position and orientation information that is gathered for the central controller will certainly not be optimized and most likely will be ill-defined. Given that each AP could potentially have a varying frame of reference, and an imprecise knowledge of its own location - the ability to measure LOS AoA components is useless in localizing an interfering radio.

To overcome the calibration problems associated with a large scale deployment of APs, *PinPoint leverages the LOS AoA measurement capability to first localize and orient the APs themselves*. By doing so, PinPoint minimizes the burden placed on the network administrator as they no longer have to ensure that all of the APs are perfectly positioned and oriented. We do assume however that we know the location and orientation of a small number (typically 3-5 per floor) of APs (referred to as anchor APs), either via GPS or manual calibration by the network ad-

ministrator. Note that this requirement is not unusual and is relatively easy to satisfy, for example, it is possible to localize a few APs that are near a window with GPS . Further, this is a one-time requirement at the time of installation, and does not require repeated surveying unlike some prior techniques [12, 13, 14].

To demonstrate how a typical enterprise network would be calibrated, we consider a deployment which consists of n APs. We assume that l of the APs (the anchor APs) already know their locations and orientations. We define $\mathcal{N}(i)$ as the set of neighboring APs within the detection range of the i^{th} AP. Each AP is equipped with ULA and has ability to measure AoA α_{ij} relative to its own axis from neighboring APs in $\mathcal{N}(i)$. The orientation h_i of each AP is the angle made by its axis to the x-axis. If the estimate of the orientation and location $[h_i\ x_i\ y_i]$ of the i^{th} AP and the location of the j^{th} AP $[x_j\ y_j]$, where $j \in \mathcal{N}(i)$, are known, estimate of the AoA $\bar{\alpha}_{ij}$, can be computed as $\bar{\alpha}_{ij} = \Psi([h_i\ (x_j - x_i)\ (y_j - y_i)]^T)$. Where the function $\Psi : \mathbf{R}^3 \rightarrow \mathbf{R}$ computes angle formed by the vector $[x\ y]^T$ with the axis of an ULA located at the origin that has orientation h.

We form penalty $\Phi(\alpha_{ij} - \bar{\alpha}_{ij})$ for each AP pairs i and j that are in communication range of each other. Here $\Phi : \mathbf{R} \rightarrow \mathbf{R}$ is a penalty function of the form $\Phi(u) = |u|^p$,where $p \geq 1$ [3, §6.1.2], for the residue between the measured angle α_{ij} and the estimated angle $\bar{\alpha}_{ij}$.

The i^{th} AP can also compute cyclic RSSI p_{ij} for signal arriving from the neighboring j^{th} AP, where $j \in \mathcal{N}(i)$. Given distance d_{ij} between the i^{th} and the j^{th} AP, the cyclic RSSI can be computed using standard path loss model as $\bar{p}_{ij} = \beta_i - 10\gamma_i \log d_{ij}$. Where β_i is a constant that is dependent on the environment of the i_{th} AP and γ_i is the path loss exponent. Since exact distance d_{ij} is not known a priori, we form penalty $\Phi(p_{ij} - \bar{p}_{ij})$ for each AP pairs i and j that are in communication range of each other.

To find the location of APs, we solve the following optimization problem,

$$\begin{array}{ll} \text{minimize} & \sum_i (\sum_j \Phi(\alpha_{ij} - \bar{\alpha}_{ij}) + \lambda \sum_j \Phi(p_{ij} - \bar{p}_{ij})) \\ \text{subject to} & [h_{n+k}\ x_{n+k}\ y_{n+k}] = [c_k\ a_k\ b_k], k = \{1, \ldots, l\} \\ & \Psi([h_i\ (x_j - x_i)\ (y_j - y_i)]^T) = \bar{\alpha}_{ij}, \\ & \bar{p}_{ij} = \beta_i - 10\gamma_i \log d_{ij}, \\ & i = \{1, \ldots, n+l\}, \qquad j \in \mathcal{N}(i) \end{array} \tag{9}$$

where the variables are x, y, h, β, γ with dimension $\mathbf{R}^{n+l}$. The problem data a, b and c with dimension $\mathbf{R}^l$ are the known x-location, y-location, and the orientation of the anchor APs. And the data α_{ij} and p_{ij} are the AoA and cyclic RSSI measurements by each APs. The above optimization problem is non-convex therefore we solve it approximately using Sequential Convex Programming (SCP) [26]. At each iteration of SCP we will fit the non-convex function Ψ and $\bar{p}_{ij}$ to some convex function within a trust region and then solve the resulting convex optimization problem to obtain a locally optimal solution. At the end of each iteration step, trust region will be updated and the convergence of the algorithm will be evaluated.

5 Interference Localization

Once the APs have been calibrated, the respective locations and orientations of every AP in the network is known. Localizing an interfering radio is now relatively straightforward. PinPoint leverages its knowledge of the signal type, direct line of sight AoA and cyclc RSSI to localize sources of interference.

In order to localize an interfering radio, PinPoint relies on the local measurements from APs near the source of interference. These APs measure the LOS AoA and cyclic RSSI and send the measurement results back to a central server. The server then aggregates the data, averages it over time to weed out noisy measurements, and triangulates the source of interference with the following optimization problem to find the location of the target radio.

$$\begin{array}{ll} \text{minimize} & \sum_j \Phi(\alpha_{jm} - \bar{\alpha}_{jm}) + \lambda \sum_j \Phi(p_{jm} - \bar{p}_{jm} \\ \text{subject to} & [h_j\ x_j\ y_j] = [c_j\ a_j\ b_j], \\ & \Psi([h_j\ (x_m - x_j)\ (y_m - y_j)]^T) = \bar{\alpha}_{jm}, \\ & \bar{p}_{jm} = \beta_j - 10\gamma_j \log d_{jm}, j \in \mathcal{N}(m) \end{array} \tag{10}$$

where the interference radio whose location $[x_m\ y_m]^T$ has to be estimated is seen by $N_m = |\mathcal{N}(m)|$ APs. x, y with dimension $\mathbf{R}^{N_m+1}$ and $h \in \mathbf{R}^{N_m}$ are the optimization variables. The problem data c, a, b with dimension $\mathbf{R}^{N_m}$ are the estimated orientations and locations of the APs that detect the interference radio. Although this problem formulation is similar to the problem (9), the size of optimization variable in this case is much smaller than the size of the optimization variable in problem (9). As a result, interfering radio sources can be localized within seconds, enabling network operators to quickly diagnose and troubleshoot sources of interference within their networks.

6 Experimental Evaluation

In this section, we evaluate the localization accuracy of PinPoint in an indoor testbed and determine how different factors such as calibration offsets, signal SNRs, and overlapping sources of interference effect performance. Below we first summarize our findings:

- PinPoint is robust and accurate, it's median error is 0.97m, around three times lower than the 3.35m and 2.94m median error for RSSI and MUSIC-AoA

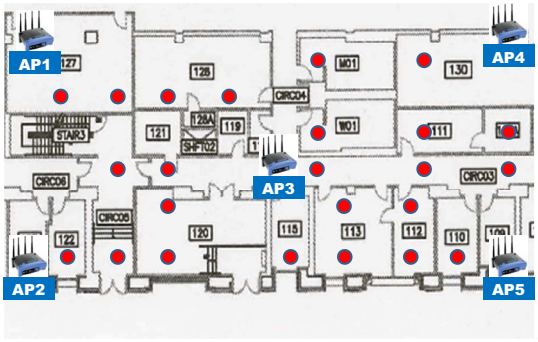


Figure 6: **Testbed Setup for PinPoint Experiments:** 4 of the APs were situated in locations where GPS signals were obtainable, and 2 among them are randomly selected as anchor nodes were selected for each experiment. Red circles indicate potential locations where interfering WiFi, Bluetooth and Zigbee radios are placed.

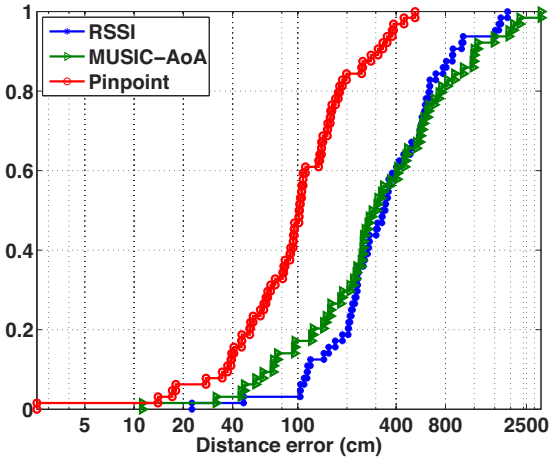


Figure 7: **Overall CDF of Localization Error for PinPoint, RSSI, and MUSIC-AoA:** PinPoint achieves a median error of 0.97 meters while both the RSSI and MUSIC-AoA only manage median errors of 3+ meters, and are worse in the 90th percentile - with errors of 10+ meters.

based approaches in our testbed across all scenarios (with and without interference). Further, PinPoint is significantly better in the tail, its 80^{th} percentile error is approximately 1.75m whereas for the RSSI and MUSIC-AoA based approaches it can be as high as 7m. Note that PinPoint can localize even though it has no information about the target radio, such as protocol, transmit power, spectrum used, data formats etc. Several prior RSSI based approaches require such a priori information to work accurately.

- PinPoint is even more accurate when there is no interference (median error of 0.83m), whereas the RSSI and MUSIC-AoA based approaches achieve 2.32m and 3.06 respectively. Thus even though PinPoint is designed to localize interference, it provides a general and accurate localization technique for all scenarios. In scenarios where there is interference, PinPoint's median error is 1.05m, while the RSSI approach worsens to a median error of more than 4m because it is unable to accurately measure RSSI under interference.
- PinPoint achieves its high accuracy assuming typical WiFi AP densities (we used the same deployment locations as the ones used by our WiFi network manager). Further, we found that even if AP density is reduced, i.e. instead of the 5 APs used to cover the full floor of 15000 sq. ft. we use only 3 APs, PinPoint can still localize accurately achieving a median error of 1.76m.

Compared Approaches: We compare PinPoint against the state of the art RSSI based approach [8, 11, 14]. Further, to make a fair comparison, we allow the RSSI based approach to know the interferer transmit power, even though in practice this may be hard to achieve since the interfering radio could be using a different modulation format (e.g. Bluetooth, Zigbee) and whatever transmit power it is capable of without the AP knowing it. Note that we do not compare against any approach that requires modifying the clients since one of our design goals is to make our technique work with legacy clients. Neither do we compare against any approach that requires extensive RF fingerprinting of the environment since our design goal is to allow quick and one-time deployment of the system.

Second, we also do not compare directly against a recently proposed approach that uses AoA measurements [15] . This technique uses a modified version of the MUSIC algorithm [9] to compute all the AoAs of signals at an AP, and then runs a heuristic to compute the location of the radio after collecting measurements from multiple APs. However, the published prior work assumes APs equipped with 8 antennas. In this evaluation however we equip APs only with 4 antennas because, in our opinion we do not see WiFi APs with more than 4 antennas being widely available and deployed, most new deployments over the next few years are expected to be with 4 antenna APs. This is due to two reasons, first MIMO throughput benefits are marginal beyond 4 antennas [7] and second the space occupied by an antenna is a major concern in many large scale deployments (e.g. an 8 antenna AP would span at least 3-4 feet assuming half wavelength spacing in the ISM band).

Setup: We evaluate PinPoint in the testbed environment shown in Fig. 6 which covers one floor of our department building and spans nearly 15000 sq. ft. We checked with our network manager the number of APs he would deploy for such a setting, and used the number he suggested (5 APs) as our baseline. Five APs to cover one floor is a common number and thus represents typical AP density. Of these 3 of the APs are manually localized and calibrated, while the rest of the APs are calibrated using PinPoint's self initialization algorithm. We also hand measure every location and orientation to determine the ground truth, however these are not used to perform

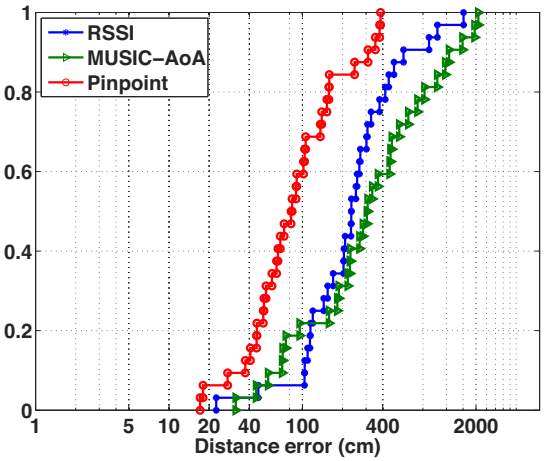


Figure 8: **Comparison of Localization Results without Interference:** PinPoint's ability to identify the LOS component's AoA helps mitigate the impact of multipath, improving performance relative to MUSIC-AoA and RSSI even when additional interference does not exist.

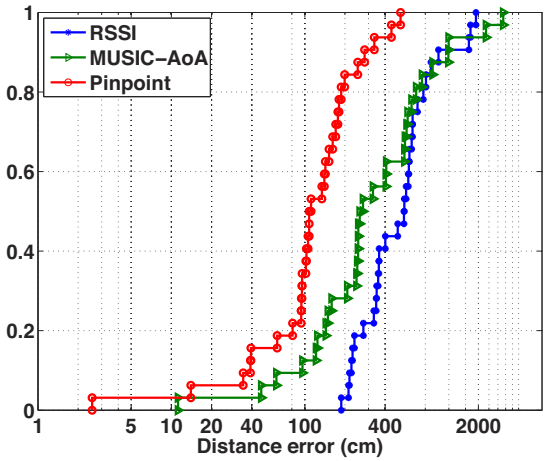


Figure 9: **Comparison of Localization Results with Interference:**.PinPoint's ability to utilize the cyclic RSSI enables it to discriminate between the signal strengths of different interfering sources, allowing it to maintain its performance even when there are multiple sources of interference.

the actual localization unless mentioned otherwise. Three types of radio interferers (802.11g, Bluetooth, 802.15.4 ZigBee) are placed at random static locations within the testbed and transmit with bursty traffic patterns which are representative of typical operation. Traces are gathered at each AP and the aggregate data is processed for localization.

6.1 Interference Localization Results

Overall Localization Performance: We start by examining the overall localization error that PinPoint achieves. For all experiments in this section, in each trial we attempt to localize one of the three radios (802.11g, Zigbee and Bluetooth) that are randomly placed in the testbed. Note that all of them could be transmitting concurrently, and other WiFi interference from the department network may also be present. Fig. 7 plots the CDF of errors for all of the interference localization trials. The curves show the performance for the three compared techniques - PinPoint, RSSI, and MUSIC-AoA.

PinPoint can localize an interfering radio to within a median error of 0.97 meters, the RSSI and MUSIC-AoA approaches can only manage median errors of 3.35 and 2.94 meters respectively, i.e. at least three times worse than PinPoint. There are two reasons for PinPoint's accuracy. First, PinPoint is inherently more accurate since it can disentangle and infer the LoS component's AoA even in severe multipath environments. Second, it is able to disentangle the target radio's signal and infer its CSSI even when there are other concurrent interfering transmissions. Neither the RSSI or MUSIC-AoA based approach possess these features.

To show that PinPoint's benefits are not primarily derived from its ability to disentangle the target radio's signal from interference, we show the performance of all three approaches when only the target radio is transmitting and no other concurrent transmissions are present. Fig. 8 shows the results. PinPoint has the best accuracy of 0.83m, while the RSSI and MUSIC-AoA approaches exhibit median errors of 2.32 and 2.98 meters respectively. The reason is PinPoint's ability to identify the LoS component's AoA, which the other two techniques do not possess and consequently their performance suffers in the harsh multipath environments that we find in indoor deployments. We therefore believe that even though PinPoint's initial design motivation was to localize interference for network management, it is a general localization technique that can be applied in a wide variety of scenarios to different applications.

In Fig. 9 we plot the performance of the three techniques with one additional interference source transmitting concurrently with the target source. PinPoint maintains sub-meter accuracy, while the RSSI approach performs poorly (median error of 4 meters and often tail errors as high as 15 meters). The MUSIC-AoA approach is less sensitive, its median error stays near 2.9 meters. The RSSI approach suffers because it cannot accurately measure RSSI of the target radio's signal alone under interference. The MUSIC-AoA approach uses the MUSIC algorithm which is robust to interference when it comes to computing the AoAs, and therefore maintains its performance.

Effect of AP Density: Intuitively, AP density affects localization accuracy since more measurements help mitigate the effects of uncertainty in the LoS AoA and CSSI measurements from individual APs. Fig. 10 plots the impact of AP density which we vary by reducing the number of APs in the testbed. As expected the median error increases as fewer APs are deployed to 1.76m when 3 APs are used to cover the entire floor. We note that this accuracy is still better than the RSSI and AoA approaches with

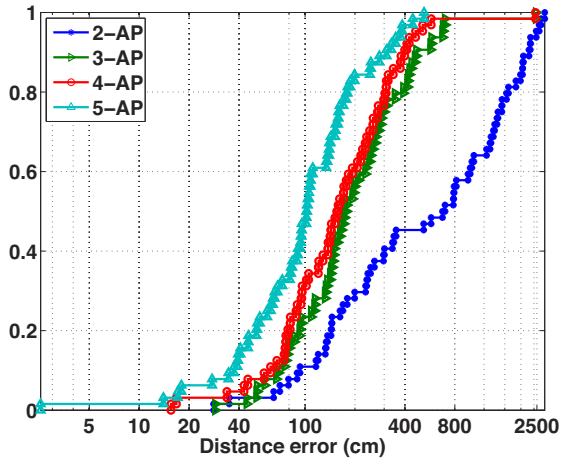


Figure 10: **Impact of AP Density on PinPoint Localization:** PinPoint performs well even in sparse AP deployments, achieving a median error of 1.76m even when only 3 APs are used to cover the entire floor.

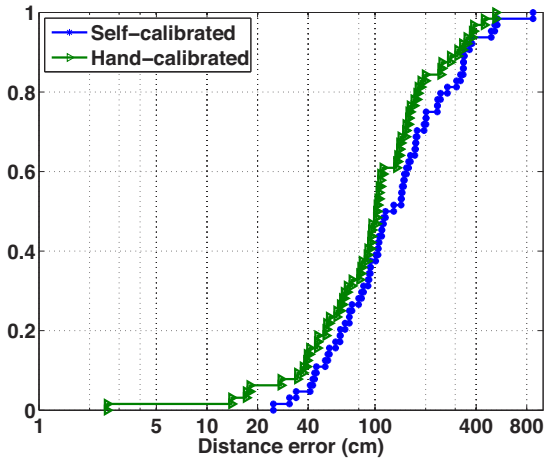


Figure 11: **Comparison of Localization with Different Calibration Procedures:** PinPoint's localization performance with self-calibrated AP locations and orientations performs just as well as PinPoint's localization performance when it is optimally calibrated by hand.

5 APs. At 2 APs however, the error is significantly worse (around 6m). As a rule of thumb, and this agrees with intuition, we found that target radios need to be visible to at least three APs to achieve good accuracy.

Impact of AP self-calibration: In Fig. 11 we measure the impact of PinPoint's AP self calibration technique on overall localization error. Specifically, we allow each AP to know its ground truth location and orientation and then compute the overall localization error for interfering radios. As we can see, there is virtually no difference in the median error, the difference is less than 5 centimeters. PinPoint's AP self-calibration performs well enough to provide very good performance that is close to the case when all APs are manually calibrated.

6.2 Performance of LOS Identification

Next, we examine how well PinPoint can disentangle the LoS component's AoA from multipath and interference. As we discussed in the design, this process has two steps: first the relative delay and angle for several packets are determined. Next, PinPoint determines whether or not a LOS component actually exists and determines how reliable the estimate actually is by using clustering techniques. If the LOS component is too weak to reliably detect, based on how large the standard deviation of the LoS cluster is, PinPoint discards the measurement so that it does not skew the subsequent localization if it is not a direct LOS path. If a sufficiently strong path exists, then it estimates the AoA and the CSSI measurement. We evaluate the accuracy of the AoA measurement alone, since there is no way of knowing the ground truth CSSI measurement reliably because it varies with time for every measurement.

Method: In this experiment, we statically place a single source of WiFi 802.11 interference within an indoor office environment. The interfering source transmits continuously at a constant average power. Measurements are then performed at various locations within the office, with the locations selected in such a way that different types of propagation paths from the source to the APs are tested.

In order to determine the ground truth for the LOS propagation path, we equip each source and receiver with compasses and annotate the placement of each with respect to landmarks in the office (wall corners, poles, etc.). We then calculate the AoA that the direct LOS path should traverse from the interference source to the receiver and use it as the benchmark for our algorithm.

Compared Approach: We compare PinPoint's LOS AoA identification against the standard algorithm used to measure AoAs, the MUSIC algorithm [9]. Since MUSIC computes all AoAs and cannot explicitly compute the LoS path's AoA, the heuristic we use is that the component with the strongest signal is the LoS AoA for MUSIC. Clearly this will not work in many scenarios, but this is the best heuristic we could come up with for comparison since it will be accurate when a strong LoS path exists.

Analysis of AoA Estimation: First, we show in Fig. 12 the CDF of the estimation error across all experimental runs. We can see that PinPoint's LOS detection achieves an accuracy of $\pm 20^\circ$ more than 65% of the time, significantly outperforming MUSIC. Notice that while PinPoint's performance degrades gracefully, MUSIC's performance drops sharply at a certain point (e.g. at the 70% mark on the CDF).

AoA Estimation in LOS vs. NLOS scenarios: In order to dive a little deeper, Fig. 13 plots the data from the previous graph in two separate groups differentiated by whether a dominant LOS path is present (solid lines) or not (dotted lines). When there is an obvious physical LOS component with no obstruction between the interfering source and the receiver, we can see that both algorithms perform quite similarly. But even in these sce-

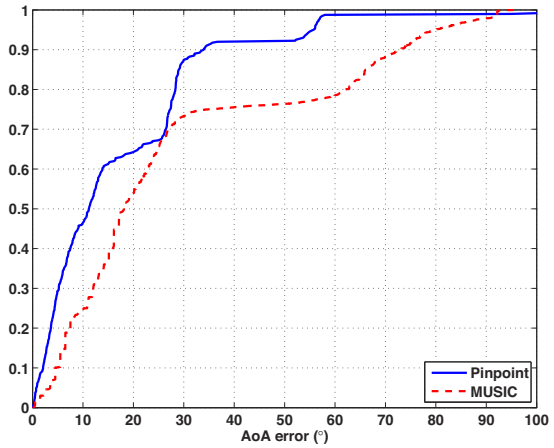


Figure 12: **Overall CDF of AoA error for PinPoint and MUSIC:** PinPoint's ability to detect the LoS AoA is demonstrated as its median AoA error is more than 25 degrees better than MUSIC.

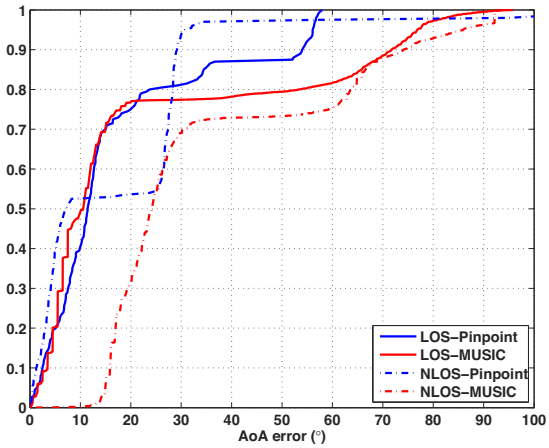


Figure 13: **AoA Error breakdown for LOS and NLOS Scenarios:** PinPoint is still able to identify the LOS component reliably when it is $> 10dB$ weaker than the strongest multipath reflection, while MUSIC's performance suffers in NLOS scenarios from locking onto stronger multipath components.

narios where the LOS component is the dominant path, MUSIC sometimes locks onto the weaker multipath component, causing sharp drop offs in performance which are seen at the tails of Fig. 13

The dotted lines in Fig. 13 show the performance of PinPoint and MUSIC when the LOS is physically obstructed. While the performance of PinPoint is only slightly worse, MUSIC is unable to correctly identify the LOS path's AoA because the secondary multipath reflections become stronger than the direct LOS path. Their performance degrades rapidly and are unable to reliably detect the LOS AoA.

7 Related Work

RSSI modeling based systems like EZ [8] assume that they get GPS locations from the users while they are walking. This training data consisting of RSSI measurements is collected at various points with a hand held mobile device across different points in the floor plan and is used to create a RSS model of the entire network. EZ achieves a median error of 2m. Another approach WiFiNet [11] also uses the RSSI for localizing the source of interference. This approach uses the off the shelf hardware to build the localization system and achieve errors of <4m. Both of these are the most recent and the best performing approaches based on RSSI modeling, other prior such approaches include [20] [21] [22] [23]. While these RSSI based methods have the attractive property of being simple and deployable on current WiFi APs, they cannot localize interference and neither are they accurate due to the inherent inaccuracy of standard RSSI as a predictor of physical distance in a rich indoor multipath environment with interference.

Other RSSI based localization systems like HORUS [12] , RADAR [13], and PINLOC [14] require significant pre-deployment effort in RF-fingerprinting. HORUS achieves median error of 0.7m, RADAR achieves median error of 1.3 m and PINLOC achieves localization granularity in 1mx1m box with 90% accuracy. All such approaches rely on the precomputed fingerprint which can become obsolete if location of some of the APs changes or if the environment changes. Fingerprinting is time consuming and expensive and has to be done periodically. Pinpoint requires no fingerprinting, and lightweight calibration of a few anchor APs at deployment. Since Pinpoint can self-calibrate the remaining APs, any changes in AP locations or the environment can be easily handled. Thus PinPoint is easy to deploy and maintain.

A recently proposed AoA based localization algorithm [15] achieves high accuracy of 0.36 m using ULA with 8 antennas. The algorithm weights the received AoA (calculated with a variant of MUSIC) by the power of the received signal. In [15], they cannot distinguish the LOS or NLOS component of the received signal, and therefore might suffer in low SNR NLOS scenarios as we saw in Sec. 6. Further, these techniques require 8 antennas at each AP, which is unrealistic for standard WiFi deployments. Other examples of the AoA based techniques are [24] [25] but these share the same shortcomings as above and generally do not provide good accuracy.

8 Conclusion

PinPoint's design highlights how one can solve interference localization tasks by leveraging the rich information hidden in RF signals. This paper designs novel signal processing algorithms and applies them to solve practical systems problems. We believe the RF signals flying around us can be mined for many more practical applications, including mapping, context detection and so on, and our future work aims to explore novel signal processing algorithms to build such applications.

References

[1] W. Gardner. Exploitation of spectral redundancy in cyclostationary signals. Signal Processing Magazine, IEEE, 8(2):14 36, apr. 1991.

[2] S. Hong, S. Katti, "DOF: A Local Wireless Information Plane", In *ACM SIGCOMM*, 2011

[3] S. Boyd and L. Vandenberghe. Convex Optimization. Cambridge University Press, 2004.

[4] L. Doherty, K. S. J. Pister, and L. E. Ghaoui. Convex position estimation in wireless sensor networks. IEEE Transactions on Communications, 2001.

[5] Bernard H. Fleury, Martin Tschudin, Ralf Heddergott, Dirk Dahlhaus, Klaus Ingeman Pedersen, "Channel Parameter Estimation in Mobile Radio Environments Using the SAGE Algorithm", In *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, 1999

[6] WARP Platform. *http://warp.rice.edu/trac/wiki/WARPLab/Downloads*, 2012.

[7] N. Czink, et. al, "Cluster Characteristics in a MIMO Indoor Propagation Environment", IEEE Transactions on Wireless Communications, 2007.

[8] K. Chintalapudi, et. al, "Indoor Localization Without the Pain", *Mobicom*, 2010.

[9] Schmidt, R.O, "Multiple Emitter Location and Signal Parameter Estimation," IEEE Trans. Antennas Propagation, Vol. AP-34 (March 1986)

[10] V. Manimohan, et.al, "Direction Estimation Using Conjugate Cyclic Cross-Correlation: More Signals than Sensors", In *ICASSP*, 1999.

[11] S. Rayanchu, et. al, "Catching Whales and Minnows using WiFiNet: Deconstructing Non-WiFi Interference using WiFi Hardware", In *NSDI*, 2011.

[12] M. Youssef, et. al, "The Horus WLAN location determination system", In *ACM MobiSys*, 2005.

[13] P. Bahl, et. al, " RADAR: An Inbuilding RF-based User Location and Tracking System", In *INFOCOM*, 2000.

[14] S. Sen, et. al, "Precise Indoor Localization using PHY Layer Information", In *HOTNETS*, 2011.

[15] J. Xiong, et. al, "Towards Fine-Grained Radio-Based Indoor Location", In *HotMobile*, 2012.

[16] R. Want, et. al, "The Active Badge Location System", In "ACM Transactions on Information Systems", 1992.

[17] N. B. Priyantha et. al , "The Cricket Location-Support System. In *MobiCom*, 2000.

[18] A. Ward et. al, "A New Location Technique for the Active Office", In *IEEE Per. Comm.*, 1997.

[19] L. Ni et. al ," LANDMARC: Indoor Location Sensing Using Active RFID", In *WINET*, 2004.

[20] Y.Gwon et. al, "ErrorCharacteristicsandCalibration-Free Techniques for Wireless LAN-based Location Estimation", In *Mobiwac*, 2004.

[21] H. Lim et. al, "Zero Configuration Robust Indoor Localization: Theory and Experimentation" In *Infocom*, 2006.

[22] Y. Ji et. al ," ARIADNE: A Dynamic Indoor Signal Map Construction and Localization System", In *MobiSys*, 2006.

[23] D.Madigan et. al, "Bayesian Indoor Positioning Systems", In *Infocom*, 2005.

[24] D. Niculescu et. al, "Ad-Hoc Position System", In *IEEE Globecom*, 2001.

[25] D. Niculescu et. al, "Ad-Hoc Positioning System (APS) using AOA", In *Infocom*, 2003.

[26] S.Boyd. "Sequential convex programming." "Lecture Notes for EE364b: Convex Optimization II, Spring Quarter 2010-11". Available at *http://www.stanford.edu/class/ee364b/lectures/seqslides.pdf*, 2011.

SloMo: Downclocking WiFi Communication

Feng Lu, Geoffrey M. Voelker, and Alex C. Snoeren

Department of Computer Science and Engineering
University of California, San Diego

Abstract

As manufacturers continue to improve the energy efficiency of battery-powered wireless devices, WiFi has become one of—if not the—most significant power draws. Hence, modern devices fastidiously manage their radios, shifting into low-power listening or sleep states whenever possible. The fundamental limitation with this approach, however, is that the radio is incapable of transmitting or receiving unless it is fully powered. Unfortunately, applications found on today's wireless devices often require frequent access to the channel.

We observe, however, that many of these same applications have relatively low bandwidth requirements. Leveraging the inherent sparsity in Direct Sequence Spread Spectrum (DSSS) modulation, we propose a transceiver design based on compressive sensing that allows WiFi devices to operate their radios at lower clock rates when receiving and transmitting at low bit rates, thus consuming less power. We have implemented our 802.11b-based design in a software radio platform, and show that it seamlessly interacts with existing WiFi deployments. Our prototype remains fully functional when the clock rate is reduced by a factor of five, potentially reducing power consumption by over 30%.

1 Introduction

Smartphones and other battery-powered wireless devices are becoming increasingly popular platforms for all manner of network applications. As a result, the energy usage of the radios on these devices is a source of considerable concern. Unsurprisingly, a large number of techniques have been proposed to help manage the power consumption of both cellular and WiFi devices. Focusing particularly on the WiFi domain, the basic approach has been to implement extremely low-power listening or sleep modes, and transition the devices into operational mode as little as possible [12, 18, 27]. The fundamental limitation with such approaches, however, is that the radio is incapable of transmitting or receiving unless it is fully powered. Unfortunately, recent studies have shown that a wide variety of popular applications make frequent and persistent use of the network [21], frustrating attempts to keep the WiFi chipset in a power-efficient state.

Transitioning in and out of sleep mode adds significant overhead, both in terms of time and energy. In particular, in addition to the costs associated with powering up the transceiver, once awake the WiFi chipset still needs to participate in the CSMA channel access scheme which frequently results in the device spending significant time in idle listening mode waiting for its turn to access the channel [18, 39]. Moreover, once a device is done transmitting or receiving, it will remain in a tail state for some period of time in anticipation of subsequent transmissions [18, 21]. To amortize these costs, the 802.11 PSM specification has nodes wake up at the granularity of the 100-ms AP beacon interval when they do not have packets to transmit. (Indeed, the popular Nexus One wakes up only every 300 ms [18].) Hence, while useful for bulk data transfers [12] or situations where traffic pattens can be predicted precisely [24], PSM-style power saving approaches are often ineffective for applications that need to send or receive data frequently [39].

In this paper, we consider an alternative to the traditional on/off model. Instead, we explore a technique that reduces the power consumption of the WiFi chipset across all of its operating modes: i.e., not just sleep and listen, but send and receive as well. Our approach leverages the excess channel capacity provided by many WiFi networks when compared to the bandwidth demands of most smartphone applications. Traditionally, when faced with low-demand clients, system designers have used excess channel capacity to improve reception rates by introducing redundant coding and/or reducing transmission power. For example, 802.11n specifies a wide variety of link rates, ranging from 1 to 150 Mbps and beyond. The lower link rates use more robust encoding and signaling schemes that can be decoded at lower signal-to-noise ratios (SNRs). These schemes translate into longer range or the ability to decrease transmission power which, along with the potential for power savings at the *sender*, can increase spatial reuse. We observe that one can instead turn excess channel capacity into an opportunity to save power at the *receiver*.

Our power savings comes from operating the WiFi chipset at a lower clock rate. Zhang and Shin demonstrated a wireless receiver that can be downclocked yet still detect packets [39]. We show how to allow transceivers to remain downclocked during reception and transmission as well. We propose a receiver design based on recent advances in compressive sensing [33] that takes

advantage of the inherent sparsity of the Direct Sequence Spread Spectrum (DSSS) modulation used by 802.11b. With our design, clients with low demand can operate their radios at a reduced clock rate while continuing to communicate with commercial WiFi devices.

We have implemented a prototype of our 802.11b-based design, called SloMo, in the Sora software radio platform. We show that SloMo seamlessly communicates with multiple vendors' commercial chipsets using standard 802.11b frames. Our measurements of frame reception rates demonstrate that SloMo remains fully functional even when the clock rate is reduced by more than a factor five. Our trace-based simulations across a range of popular smartphone applications show that SloMo reduces WiFi power consumption by up to 30–34% on the iPhone 4S and Nexus S, respectively. Moreover, SloMo outperforms two other proposed approaches, U-APSD and E-MiLi, in almost all cases.

2 Related work

There has been a great deal of work on improving the energy efficiency of WiFi devices. These efforts can be broadly classified into three categories: 1) improvements to 802.11 PSM, 2) systems that duty cycle the WiFi device, and 3) attempts to decease transmit power.

Efficient power save modes. Most approaches rely on placing the device in a low-power sleep mode whenever possible. The two basic alternatives are to coordinate these periods of sleep between the access point and the device, either through periodic polling (as with the 802.11 PSM standard) or deliberate scheduling [27]. Others have proposed dynamically adjusting sleep periods based upon a client's traffic pattern [2, 16]. Researchers have previously noted the disparity between modern 802.11 link speeds and the traffic demands of many clients. μPM suggests powering down low-demand WiFi clients between individual frame transmissions [17], relying upon 802.11 devices retransmitting unacknowledged frames to limit losses. Catnap [12] extends this approach by estimating bottleneck throughput and scheduling client wake-ups based upon the predicted availability of data from the wide-area network.

One challenge with these approaches is that, when awake, a WiFi device must participate in the channel contention process. Studies have shown that this process can consume considerable amounts of energy, especially in dense deployments where nodes are in range of multiple APs. SleepWell coordinates sleep cycles among neighboring APs to decrease contention during wake-ups, thereby increasing client power efficiency [18].

Finally, even otherwise-effective power saving mechanisms implemented by the WiFi chipset can be overridden by applications in many popular frameworks [4, 5]: some apps prevent the WiFi device from entering PSM mode, forcing the WiFi card to stay awake in an effort to improve performance [9, 35]. Because SloMo decreases power consumption across all WiFi states, it can still reduce energy consumption in these cases.

Device duty cycling. Others take a more drastic approach: rather than entering low-power sleep modes, they identify times when it is possible to simply turn a WiFi device off entirely. One early system, SPAN [7], turns off entire nodes in multi-hop ad hoc wireless networks if the connectivity of the network can be preserved without them. In more general environments, systems have been designed to keep WiFi powered down by default, and use an out-of-band signal to asynchronously alert the device of pending data [1, 31]. Since smartphones may frequently be outside the coverage area of a WiFi AP, the only reason to keep the WiFi transceiver powered is to determine when coverage returns. Many systems have attempted to reclaim this energy by instead duty cycling WiFi radios based upon predictions of WiFi availability. These predictions are variously based upon the detection of nearby Bluetooth devices [3] or cell towers [26], or historical device movement patterns [20].

Limited transmit power. Finally, a direct approach to decreasing WiFi power draw while transmitting is to reduce radiated energy. WiFi transceivers can leverage transmit power control to emit signals using sub-mW energy when the SNR is high. Unfortunately, despite the obvious attractiveness of such an approach, studies have repeatedly shown that adjusting transmit power has little impact on the total power draw of commercial 802.11 devices due to the limited power consumption of the power amplifier relative to the rest of the electronics [15, 22].

Downclocking. We take a radically different approach by enabling the radio to communicate while in a low-power state. Our efforts are inspired by previous observations that radios can conserve power by operating at lower clock rates. Researchers have argued that devices could dynamically adjust their sampling rate based upon the frequencies contained within the observed signal [11], but their approach is not directly applicable to the encoding schemes employed by WiFi. In the context of WiFi, recent proposals argue that next-generation systems should support multiple channel widths and adapt their instantaneous channel width based on the offered load [6] (although stations operating in different bandwidths cannot decode each other's transmission and the 17-ms switching overhead makes co-existence challenging), and develop mechanisms to detect packet arrivals in a downclocked state [39]. Downclocking a receiver through dynamic frequency scaling has been applied in the wireline context in the past [29], but we are not aware of any similar schemes in the wireless domain.

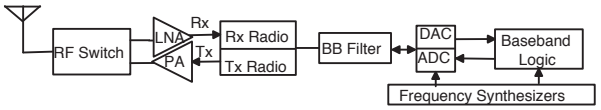


Figure 1: A simplified WiFi card architecture.

3 Motivation

As wireless link speeds continue to increase, mobile devices are increasingly likely to want to use only a small fraction of the channel capacity. With WiFi, however, use of the network is an all-or-nothing affair in terms of power: if a transceiver is not fully powered, no data can be sent or received.

3.1 The potential of downclocking

The power consumption of a CMOS computing device is proportional to its clock rate [25]. Not surprisingly, dynamic frequency scaling (DFS) has long been used as a technique to save power in a variety of computing domains [36]. Fundamentally, the same rules apply to wireless transceivers: downclocking the radio hardware can result in significant power savings. The challenge in downclocking radio equipment, however, is that the Nyquist theorem dictates that to successfully receive a signal, the receiver must sample the channel at twice the bandwidth of the signal [30]. In practice, today's WiFi devices are designed in such a way that the frequency of the entire radio pipeline is gated by the sampling rate.

Figure 1 shows a typical WiFi transceiver architecture. The analog baseband signal is first processed by a baseband filter to confine the signal to the desired band. It is then sampled by an analog-to-digital converter (ADC) and data samples are passed to the baseband processor, which decodes the signal and uploads the recovered frame to the host. The entire radio card is driven by a common crystal oscillator, which feeds the frequency synthesizer and the phase locked loop (PLL). The frequency synthesizer generates the center frequency for RF operation while the PLL serves as the clock source for the ADC and baseband processor. For a 22-MHz 802.11b channel, the radio runs at 44 MHz (or faster).

As a result, the channel sampling rate directly determines the permissible clocking rate—and power consumption—of the WiFi card. Previous studies have shown that the power consumption of popular WiFi chipsets (e.g., from Atheros and Netgear) does indeed vary with frequency [6, 39], although the precise relationship depends on what the device is doing (sending frames, receiving frames, or idling) and differs across chipsets. As an example, Table 1 shows the reported energy consumption of a popular WiFi chipset while operating at various clock rates [39].

Not surprisingly, the power savings are sub-linear (40% savings while receiving packets at a 25% clock rate), but they are still substantial. However, current devices were not designed to be downclocked. Hence, it is unlikely they are optimized to be power-efficient at frequencies other than their target operating point.

Clock rate	25%	50%	Full rate
Idle	640 mW	780 mW	1200 mW
Rx	980 mW	1440 mW	1600 mW
Tx	1210 mW	1460 mW	1710 mW

Table 1: Power draw of the Atheros 5414 WiFi chipset in the LinkSys WPC55AG NIC at various clock rates [39].

3.2 Downclocked transmission

It is not obvious that downclocking a radio would be beneficial while transmitting data: the lower the data rate, the longer the transmission takes. Hence, in theory one should transmit as fast as possible and place the radio back into low-power mode as soon as transmission is complete. Alternatively, one could realize similar savings by transmitting at a low data rate and scaling back the transmission power. These approaches, however, presume that the frequency and/or power of the transceiver can be adjusted efficiently.

Moreover, even if the device only receives data, the 802.11 specification requires that it transmit an ACK frame to confirm receipt of the data frame—and the ACK frame must be sent within a strict, 20-μs interframe time (SIFS). As with reception, Nyquist requires that the transceiver operate at twice the signal bandwidth to transmit the standard Barker sequence. While some chipsets, such as the MAXIM 2831, are able to switch back to full clock rate in time to transmit an ACK frame, others take substantially longer (e.g., an Atheros 5414 takes roughly 125 μs to switch clock rates [39]). In such cases, to realize the benefits of downclocked reception, the transceiver needs to transmit at a slower clock rate an ACK frame that a standard-compliant WiFi transmitter will accept. (The Rx power draws in Table 1 assume the device remains downclocked for ACK transmissions.)

The potential benefits of downclocked transmission go even further when considering the energy spent on clear channel assessment (CCA) when a node attempts to gain access to the channel. Previous studies have shown that CCA is the dominant power drain when there is a high contention level in the network [18, 39]. Most commercial WiFi chipsets implement the carrier sensing component of CCA, i.e., determining whether the channel is free, using energy detection, which can be conducted at virtually any clock rate. Moreover, modern WiFi cards seem to be more power proportional when in this so-called idle listening state. As shown in Table 1, the measured Atheros chipset consumes 47% less power in idle listening mode when downclocked by a factor of 4.

However, once the channel is detected to be idle, a WiFi station needs to attempt to transmit a frame within very short order (as little as 50 μs depending on its current back-off interval). Given the switching times of commodity chipsets, these timing requirements suggest that WiFi devices are likely to need to perform carrier sensing and frame transmission at the same clock rate. In other words, in order to perform CCA while downclocked, the WiFi device must be prepared to transmit while downclocked as well.

3.3 Network impacts

Clearly, downclocked nodes have the potential to realize significant power savings. An obvious concern, however, is that the lower bitrate transmissions require more airtime, thereby decreasing overall network performance, or, worse, increasing the energy consumption of other nodes in the WiFi network and negating the gains realized by the downclocked node. While certainly possible in theory—or even in practice for highly congested networks [34]—its likelihood depends both on the background usage level in the network and the communication patterns of the downclocked node.

For example, for VoIP applications the typical packet size is roughly 40 bytes, implying that a VoIP node's air time usage is dominated by inter-frame spacing and channel contention resolution rather than data transmission or reception [34]. Hence, a VoIP flow's impact on network throughput is likely to be negligible regardless of the bitrate (clock rate) the node chooses to employ. In other scenarios, however, where the downclocked node is transmitting or receiving large packets, or the network is already reaching maximum capacity, the impact may be noticeable. We observe that both the station and the AP could detect and address such situations. In particular, an AP can monitor the current traffic load on the network, number of PSM clients, and any other pertinent information. For severely congested networks, the downclocked operation may not be allowed. In practice, for most of the popular smartphone apps we have studied, the impact on free channel airtime is limited ($\leq 16\%$, see Section 6.3). Moreover, many networks are lightly loaded. For example, a study of our department's wireless network found that 60% of all frames are transmitted without contention—i.e., the initial back-off counters expire without needing to wait for other channel activity [8].

4 Downclocked 802.11b

In this section we describe the design of SloMo, our prototype downclocked radio for 802.11b. SloMo can fully interoperate with standard-compliant WiFi devices (i.e., 802.11a/b/g/n/ac) at both 1 and 2-Mbps DSSS rates, with no modifications to the access point. While these data rates are admittedly modest, we show later that they suffice for many popular applications. Further, the 802.11b rates remain widely supported in both deployed WiFi networks and the upcoming 802.11ac chipsets (e.g., Broadcom 4335) and routers (e.g., Cisco EA6500). Indeed, due to its robust communication range and low cost, 802.11b is the only supported WiFi mode in some special-purpose devices [13, 14, 37].

4.1 Reception

Our receiver design is based upon an observation that the process of direct-sequence spread spectrum (DSSS) modulation, as employed by the 802.11b standard, bares a great similarity to a recently proposed compressive sensing (CS) decoding scheme. DSSS and complementary code keying (CCK) are the two modulation techniques specified in the IEEE 802.11b standard. When the data rate is 1 or 2 Mbps, only DSSS modulation is employed. The difference between the 1 and 2-Mbps encodings lies in whether the quadrature component of the carrier frequency is used: they employ binary phase shift keying (BPSK) and quadrature phase shift keying (QPSK), respectively. To ease our explanation, we will focus our discussion on the 1-Mbps BPSK scenario; the methods can be similarly applied to 2-Mbps QPSK encoding as we demonstrate.

In their recent breakthrough, Tropp *et al.* observe that it is possible to employ compressive sensing to decode digital signals while sampling at rates far below the Nyquist rate, provided the signal is sparse in the frequency domain [33]. Their approach mixes the sparse signal they wish to decode with a high-rate chip sequence to spread its signal band. They show that in many cases the information contained in a sub-band of the resulting spread signal turns out to be sufficient for recovering the original signal.

DSSS modulation is analogous to the first stage of this process: the baseband signal is also spread over a wide range of bandwidth. Though the spreading in 802.11b is designed to increase the signal to noise ratio (SNR) at the receiver, it also provides the opportunity to apply compressive sensing by only looking at part of the band when SNR is not an issue.

4.1.1 DSSS modulation

The transmission chain of a standard 802.11b implementation can be summarized as four steps: scrambling, modulation, spreading and pulse shaping. The data is initially "scrambled" by XORing it with a fixed pseudo-random sequence—to avoid long runs of ones or zeros—before being modulated (using BPSK in the 1 Mbps case). The modulated baseband signal is then "spread" by replacing each bit with an 11-chip Barker sequence to expand the signal. The spreading process serves several

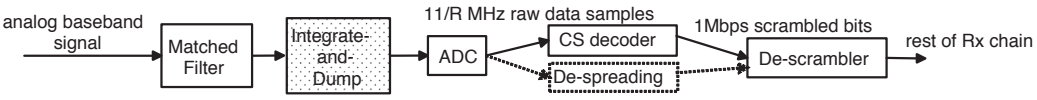


Figure 2: Original and modified baseband Rx processing chain. Compared to the original Rx chain, the modified chain adds an additional integrate-and-dump component and replaces the De-spreading part with the CS decoder.

purposes. First, it enlarges the spectrum of the original baseband signal by 11× to make it more robust to channel noise. Secondly, due to the unique properties of a Barker sequence, it enables the receiver to more easily synchronize with the transmitted signal. In particular, a Barker sequence has low auto-correlation except when precisely aligned with itself, so receivers can easily determine when they have correctly synchronized with the incoming chip sequence.

Mathematically, one can consider the DSSS spreading process as computing an 11-chip signal, $\mathbf{C}$, for each bit, $\mathbf{C} = \mathbf{M} \cdot \mathbf{b_i}$, where $\mathbf{b_i}$ is a 2×1 sparse vector ($b_1 = [\mathbf{0\ 1}]^\mathbf{T}$ corresponds to a 1 and $b_0 = [\mathbf{1\ 0}]^\mathbf{T}$ for a 0), and the Barker sequence $\mathbf{M}$ is given by

$$\mathbf{M} = \begin{bmatrix} +1 -1 +1 +1 -1 +1 +1 +1 -1 -1 -1 \\ -1 +1 -1 -1 +1 -1 -1 -1 +1 +1 +1 \end{bmatrix}^\mathbf{T}$$

Note that the two rows of $\mathbf{M}$ are simply inverses of each other; hence, both Barker sequences have identical auto-correlation magnitudes—they just result in either positive or negative correlation.

Subsequently, the pulse shaping stage ensures that the resulting signal spectrum shape conforms to the IEEE 802.11b specification. In particular, the shaped signal has a bandwidth of 22 MHz; therefore, a minimum sampling rate of 44 MHz is required to meet the Nyquist sampling criteria at the receiver side.

Conversely, Figure 2 presents a high-level description of an 802.11b receiver baseband processing chain. A matched filter recovers the chip values. In particular, the matched filter correlates the incoming chip samples with the Barker sequence to locate where the bit boundary is, i.e., the first chip in the bit. Once the signal is synchronized, it is sampled every chip time. Therefore, over the course of a single bit duration, 11 sample values will be collected corresponding to the 11-chip Barker sequence. This chip sequence is "de-spread" by once again correlating it with the Barker sequence to determine whether a 1 or 0 was encoded, resulting in (hopefully) the original 1-Mbps bit stream which is then de-scrambled by XORing with the same scrambler sequence.

4.1.2 Compressive sensing

We implement compressive sensing using an integrate-and-dump sampler as suggested by Tropp *et al.* [33]. We extend the match filter by introducing an integrate-and-dump stage, which accumulates the output from the matched filter for multiple chip durations, allowing for a lower sampling rate than the standard 11 MHz. The radio can then be downclocked appropriately to achieve a desired compression ratio: sampling is performed on the accumulated output (as opposed to each chip) and the discrete samples—which contain multiple chips—are fed to the rest of the receiver chain.

We can formalize the DSSS sampling process described in the previous subsection as extracting a sample $\mathbf{Y}$ from the received signal, $\tilde{\mathbf{C}}$ (which is the transmitted DSSS signal $\mathbf{C}$ encoded as described above but distorted by the channel), with the diagonal sampling matrix $\mathbf{H}$:

$$\mathbf{Y} = \mathbf{H}\tilde{\mathbf{C}}. \quad (1)$$

In a standard receiver operating at full clock rate, $\mathbf{H}$ is an 11×11 identity matrix which simply samples each chip exactly once. $\mathbf{Y}$ is then correlated with the Barker sequence $\mathbf{M}$ to determine the transmitted bit.

With an integrate-and-dumper sampler, the measurements can be viewed as a linear combination of the original chip values. For example, suppose only 3 measurements are desired (i.e., a downclocking ratio of 3/11). The measurements can be viewed as substituting a compressive measurement matrix into Equation 1:

$$\hat{\mathbf{H}} = \begin{bmatrix} 1\,1\,1\,1\,0\,0\,0\,0\,0\,0\,0 \\ 0\,0\,0\,0\,1\,1\,1\,1\,0\,0\,0 \\ 0\,0\,0\,0\,0\,0\,0\,0\,1\,1\,1 \end{bmatrix}$$

Here, our sampling matrix has only three rows because we intend to sample each bit's Barker sequence only three times. Because 11 cannot be evenly divided by 3, the integrate-and-dump sampler needs to accommodate varied accumulation length. (We relax this assumption in a later subsection.) In this particular example, to reduce the clock rate by 11/3=3.67×, we choose to take two samples of 4 chips and one of 3. Once the compressive samples are obtained, the baseband logic can be re-engineered to work with the compressed measurements. For example, Davenport *et al.* show the following decision rule[1] can be used [10]:

$$d_i = (\mathbf{Y} - \mathbf{HMb_i})^\mathbf{T}(\mathbf{HH^T})^{-\mathbf{1}}(\mathbf{Y} - \mathbf{HMb_i}).$$

[1]The middle term (pre-whitened matrix) of Davenport's decision rule is actually $(\mathbf{HMM^TH^T})$ because they assume the basis matrix $\mathbf{M}$ is applied during decoding after the signal has been received. In our DSSS modulation scheme, the matrix is applied during transmission, so we can drop it from our rule.

If $d_0 < d_1$, the bit is decoded as 0, and 1 otherwise. Our proposed receiver baseband processing chain is also presented in Figure 2. Since only a single bit is decoded at a time, the decision rule can be simplified as

$$d_i = \mathbf{Y^T}(\mathbf{HH^T})^{-1}(\mathbf{HMb_i}). \quad (2)$$

4.2 Transmission

Recall from the previous section that one of the key roles of the Barker sequence is to allow the receiver's matched filter to identify the beginning of the bit sequence. In particular, given 802.11b's 11-bit Barker sequence, a bit boundary is within the next 10 samples of any chip. Hence, the matched filter simply correlates the chip samples at each of these 11 positions. Because of the auto-correlation properties of the Barker sequence, the start of the bit sequence is clearly indicated by a correlation peak. In theory, the Barker code's correlation maximum is 11× larger than the second maximum. However, when a signal is transmitted over the air, it may get distorted and noise is added. Hence, real receivers never use a peak criteria as high as 11; on the contrary, commercial WiFi cards use much lower thresholds as our experiments reveal.[2]

Based on this observation regarding the decoding threshold, we design "Barker-like" sequences whose auto-correlation properties are not as strong as regular Barker sequences, but are still likely to satisfy the matched filter's threshold to allow the receiver to properly identify the bit boundary. Similarly, our sequences have the property that, when correlated with a properly aligned 802.11b Barker sequence, they can be successfully decoded. (Recall that de-spreading is only performed on properly aligned chip sequences.) Again, they do not have perfect correlation with the true Barker sequence, but sufficiently high enough to either exceed the threshold for 1s, or low enough to pass for 0s.

The key feature of our Barker-like sequences is that they are shorter than the original Barker sequence, yet transmitted over the same time interval. As a result, each chip in our Barker-like sequence lasts longer than a standard Barker chip. The exact number of chips in the sequence—and, thus, the chip duration—can be chosen to match an intended downclock rate. We omit the detailed mathematical steps involved to search for these sequences. At a high level, we use correlation peak-to-average ratio as a close approximation to decide how good the code sequence is. Figure 3 shows some examples of the Barker-like sequences we obtain, and how they compare to the original 11-chip Barker sequence. To operate at a particular downclocked rate of $m/11$, we select a Barker-like sequence of length m to use for spreading. Because the radio is downclocked, each chip will last $(11/m)\times$ as long as a standard chip[3], and the signal will be more narrow than usual.

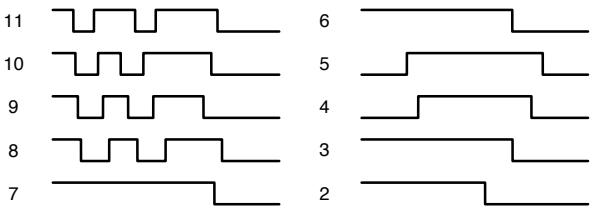


Figure 3: Spreading sequences of various lengths, including the 802.11-standard 11-chip Barker sequence and the Barker-like sequence used by SloMo when downclocked accordingly.

4.3 Practical design considerations

In the previous description of our compressive sensing based receiver, we assume 1) that the bit boundary is known, 2) compressive measurements are only taken over chips belonging to the same bit, and 3) the number of chips to be integrated varies (as reflected in the measurement matrix given in Section 4.1.2). Here, we first relax the latter two requirements and then return to address the former.

4.3.1 Fixed-length integrate-and-dump

Rather than have variable-length integration periods, an alternative is to have a fixed integration length l and occasionally integrate fractions of a chip value into a measurement. For example, the following measurement matrix (which we employ when the clock is operated at 4/11 of the original rate) serves as a concrete example:

$$\mathbf{H} = \begin{bmatrix} 1\,1\,1\,0\,0\,0\,0\,0\;0\;\;0\,0 \\ 0\,0\,0\,1\,1\,1\,0\,0\;0\;\;0\,0 \\ 0\,0\,0\,0\,0\,0\,1\,1\,0.5\,0\,0 \\ 0\,0\,0\,0\,0\,0\,0\,0\,0.5\,1\,1 \end{bmatrix}$$

Note that the third and fourth samples each integrate a fraction of the 9th chip. Alternatively, if integrating a fraction of the chip turns out to be challenging, we could integrate a fixed number of chips and extend the decoding to multiple bits group.

At a raw data bit rate of 1 Mbps, Nyquist requires a minimum of two measurements per 11-bit chip sequence; we cannot downclock the receiver a full 11×. Hence, the useful range of integration lengths is between 2 and 5 chips. Since 11 is a prime number, for any integration length k ($2 \le k \le 5$), the number of compressed samples is not an integer for a single bit. In fact, we need to perform compressive sensing over a minimum of $11k$ chips to produce an integer number of measurements

[2]For example, Sora [32] decides the maximum value is a peak if the maximum value is at least twice the second maximum.

[3]Except when $m = 2$ where the two chips last for 6 and 5×, respectively, which we found to be more reliable than two chips of 5.5×.

(i.e., 11 measurements). Therefore, rather than decoding one bit at a time, we jointly decode k bits in a group—which exactly corresponds to $11k$ chips.

4.3.2 Synchronization

Symbol synchronization is a fairly standard technique and is often implemented in hardware [23]. Unfortunately, locating the bit boundary is slightly more challenging when using compressive sensing. After passing through the integrate-and-dump circuit, the compressed measurements no longer exhibit the excellent auto-correlation property provided by the original Barker sequence. Therefore, the standard correlation and peak searching-based method described in the previous section no longer suffices.

Recall that we are decoding our sample stream in groups of k bits at a time, where each bit consists of 11 chips, but our integrate-and-dump sampler has reduced these chips by a factor of k. Hence, we are always decoding exactly 11 samples at once. If we knew where to start decoding, the first compressed measurement would correspond to the sum of the first k chips of the first bit. Rather than trying to identify the bit boundaries ahead of time, we observe that 11 is a prime number, so one and only one alignment with the sample stream will produce successful decodings—all others will never align regardless of the downclocking factor k.

Because we have no idea which one of the 11 compressed measurements starts a group, we store the decoding results for each possible position simultaneously. For implementation purposes, we keep 11 bit arrays (B_0-B_{10}). Suppose the incoming compressed measurements are labeled S_0, S_1, $\cdots$; we decode $S_0 - S_{10}$ and store the result in B_0, $S_1 - S_{11}$ to B_1, $S_2 - S_{12}$ to B_2, $\cdots$, $S_{10*j+i} - S_{10*j+i+10}$ to B_i ($0 \leq i, j, \leq 10$). Each incoming compressed measurement will complete the decoding of one of the 11 bit arrays. Meanwhile, we look for the fixed bit pattern of the Start of Frame Delimiter (SFD) among the 11 stored bit arrays. Once the SFD is identified in one of the arrays, we know the correct bit boundary and we only need to keep decoding in one of the arrays.

While the synchronization operation can be conducted in parallel, we implement the process in a single software thread in SloMo as the synchronization stage only lasts for the duration of the preamble (72 μs and 144 μs for short and long preambles, respectively). The SFD is guaranteed to be found within this well-defined time bound (or equivalently, a fixed number of decoded bits) for any valid frame. If no SFD is detected after a reasonable amount of time, the synchronization process is aborted and we start to search for the next packet.

4.4 Interacting with existing networks

Because SloMo requires modifications only to the downclocked wireless node and is entirely 802.11b-compliant, it is fully compatible with existing WiFi deployments. No changes need to be made to the access point or other devices on the network to support SloMo. In this section, we discuss how SloMo interacts with a standard 802.11b/g/n basestation, as well the potential interactions with other client nodes due to its use of 802.11b (as opposed to 11g or 11n).

4.4.1 Rate selection

When operating in downclocked mode, a SloMo node can only decode frames encoded using DSSS—in particular, it is not able to use CCK encoding (i.e., 5.5 and 11-Mbps 802.11b frames) or communicate at 802.11g/n rates. Fortunately, the 802.11b standard includes mechanisms for the SloMo node to convey these constraints to the AP. If the SloMo node is currently connected to an AP, before it goes into downclocked mode it can transmit a re-association request frame to inform the AP it only supports 1 and 2 Mbps. Even if a SloMo node fails to notify the AP of the supported rate change, most APs employ a dynamic transmission rate adjustment algorithm that will throttle the sending rate until it successfully communicates with the SloMo station: when the AP fails to receive an ACK for frames it transmits at a higher data rate, it will retry at a lower rate and eventually step down to 1 or 2 Mbps.

4.4.2 Protocol interactions

While SloMo devices must operate at 802.11b speeds, it is clearly desirable to ensure that other network nodes can continue to transmit at 11g or 11n rates if they are so capable. The concern in such environments is that the SloMo node cannot decode such frames, and might cause collisions. Luckily, collisions are straightforward to avoid. 802.11b specifies three different clear channel assessment methods: energy detection, frame detection, or a combination of the two. An 802.11b-compliant device can implement whichever method it chooses. Relying on energy detection alone as its CCA method, our SloMo node could co-exist with any other 11g/n node in the network *without* requiring them to turn on protection mode to minimize the impact of throughput loss due to slower 11b rates. This approach may require the network operator to manually turn off protection mode on the AP if SloMo nodes are the only possible set of 802.11b clients.

Additionally, because 11b and 11g employ different inter-frame timings (for example, the slot time is 20 μs and 9 μs for 11b and 11g with protection mode off, respectively), one might be concerned about the potential

unfairness in channel access contention. We could modify the inter-frame timings for SloMo nodes to ensure fair channel access, but we observe that the standard settings penalize the SloMo node, not the other nodes, and the SloMo node is unlikely to have high demand for the channel given it has elected to go into downclocked mode. Hence, we have not deployed this change on our prototype.

5 Prototype

To assess the feasibility of our approach, we implement a prototype CS-based 802.11b transceiver architecture in Microsoft Sora, a fully programmable software defined radio [32]. We show that compressive sensing achieves similar packet reception rates as standard WiFi under reasonable network conditions, even when clock rates are reduced by a factor of five. We also show that downclocked transmission using short "Barker-like" sequences is feasible when communicating with standard WiFi devices.

5.1 Implementation

To allow maximum generality of their radio platform, the Sora architecture differs from the typical WiFi chipset design discussed previously. Rather than implementing a matched filter in hardware and sampling thereafter, the Sora radio board has a fixed sampling rate of 44 MHz and passes the raw data samples directly to the processing pipeline. The matched filter and decoding stages are all implemented in software.

We modified the Sora code by adding the integrate-and-dump sampler in the receiver chain and re-design the bit decoding algorithm as described by Equation 2. We have implemented both versions of the integrate-and-dump sampler. Since Sora's clock rate is fixed at 44 MHz, we are unable to downclock it while transmitting. Instead, we emulate downclocked transmission by repeating data samples to effectively simulate a slower clock. We then employ a root-raised-cosine filter for pulse shaping. Since Sora does not have an on-board automatic gain control (AGC) circuit, we have to realize the AGC in software. Finally, to compensate for the clock oscillator difference between transmitter and receiver, we also implement the phase tracking component to ensure correct decoding of multiple-bit groups.

5.2 Experimental configuration

We conduct most of our experiments using two nodes, a Sora node running our SloMo implementation and a laptop with a commercial WiFi device. The Sora hardware is a Shuttle XPC SX58J3 machine with 8 CPU cores configured with a Sora radio control board and an Ettus Research XCVR2450 radio transceiver. It runs Windows XP modified to support the baseline Sora software and our SloMo modifications. The laptop is a Lenovo T410 with 2 CPU cores running Ubuntu 10.04 with an Intel 6200 WiFi card. We operate the Sora node and laptop as an ad-hoc network for flexibility. By default we perform our experiments using the 1-Mbps link rate of 802.11b (experiments using 2-Mbps link rates double application throughput as expected).

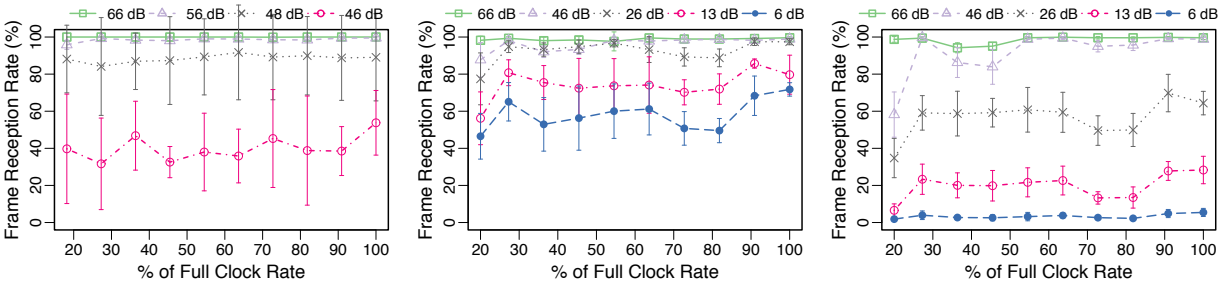
To experiment with different network conditions, we varied the distance and path between the nodes. We fixed the location of the SloMo node in a room, and moved the laptop to various locations inside the building.

5.3 Downclocked reception

We start by evaluating downclocked reception in isolation using compressive sensing (CS). We transmit packets using the commercial WiFi device on the laptop to our experimental Sora node, which receives them using CS with a configurable decoding clock rate. For each clock rate and location, we transmit 1,000 UDP packets (each 1,000-bytes long) paced to allow the network to settle between transmissions. We repeat each experiment 10 times to account for variations. We perform the experiment across a wide range of clock rates, and in different locations that result in a variety of network conditions. In each case we record the fraction of transmitted packets successfully received and decoded using CS on the Sora node, and also report the corresponding SNR value for each location.

Figure 4(a) shows that downclocked reception operates nearly as well as standard WiFi across a wide range of decoding clock rates. Each point is the average of 10 runs, and the error bars show the standard deviation. A clock rate of 100% corresponds to standard WiFi processing as the baseline, and smaller rates correspond to more aggressive use of compressive sensing with lower power requirements. When the SNR is good ($\geq$48 dB), packet reception using compressive sensing is nearly equivalent to standard WiFi, even for very low clock rates of 18–36%. Recall from Section 3.1 that downclocking at such rates corresponds to more than 40% savings in power consumption for a popular WiFi chipset.

Unfortunately, our ability to evaluate SloMo downclocked reception performance for a wider range of SNRs is limited by the Sora platform. We observe that Sora has a rather narrow dynamic range in terms of receiver sensitivity and exhibits a sharp cut-off behavior when the SNR is around 46 dB, likely due to the lack of hardware automatic gain control. While operating in this regime, Sora's standard WiFi implementation only achieves a 53% reception rate, and compressive sensing delivers 73% of that performance at the lowest clock rate.



(a) WiFi → SloMo (b) SloMo → WiFi (small packets) (c) SloMo → WiFi (large packets)

Figure 4: Frame reception rates at SloMo Sora node (commercial WiFi device) for packets sent by commercial WiFi device (SloMo Sora node) using downclocked compressive sensing reception (downclocked "Barker-like" transmission). As a baseline, the 100% clock rate corresponds to using the default 802.11b implementation.

5.4 Downclocked transmission

Next we evaluate downclocked transmission in isolation using the shorter "Barker-like" sequences. We send packets from our experimental Sora node using downclocked transmission to the commercial WiFi device on the laptop, and record the fraction of transmitted packets successfully received and decoded by the commercial device. We use the same methodology as with compressive sensing: 10 runs of 1,000 UDP packets at each combination of downclock rate and network location. We also experiment with two packet sizes. The first is a small packet size of 60 bytes, corresponding to apps sending small data packets and sending ACKs in response to a packet received using compressive sensing. The second is a larger packet size of 1,000 bytes.

Figures 4(b) and 4(c) show the results for downclocked transmission for small and large packets, respectively. Compared to downclocked reception with compressive sensing, we note that the operational SNR range is much larger; commercial WiFi cards have much better receiver sensitivity than Sora.

Focusing on results relative to the commercial WiFi baseline, however, shows that downclocked transmission using shorter "Barker-like" sequences more strongly depends on network conditions, clock rate and packet size. A clock rate of 100% transmits using the full Barker sequence in standard WiFi, and smaller rates correspond to transmission using increasingly shorter Barker-like sequences (Figure 3); the lowest transmission clock rate is 20%, which corresponds to transmitting with just two chips (Section 4.2). As shown in Figure 4(b), with small packet sizes downclocked transmission is nearly as good as standard WiFi for moderate and good network conditions ($\geq$ 26 dB) for nearly all downclock rates (at the lowest 20% clock rate, reception rates are 10–20% below the baseline). With larger packets sizes, as shown in Figure 4(c), downclocked transmission continues to do well for the majority of clock rates. Note that downclocked rates of 73% and 82% underperform other clock rates by 7–10% when the SNR is moderate or low ($\leq$26 dB). This variation is due to how well a "Barker-like" sequence approximates the original Barker sequence; a longer sequence (higher clock rate) does not necessarily yield better correlation results. As with small packets, the lowest clock rate of 20% substantially degrades reception relative to the baseline, pushing the limit of downclocking.

Overall, when SNR is poor ($\leq$ 13 dB), downclocked reception rates are on average 10% less than the standard WiFi implementation; otherwise, the packet reception rates are approximately the same. These results indicate that downclocked transmission is feasible for a wide range of SNR scenarios, especially transmitting ACKs at the same downclocked rate used to receive data frames.

5.5 Further prototype experiments

We performed additional experiments with the SloMo prototype, which we summarize for space considerations. First, we combined downclocked reception and transmission to evaluate the quality of Skype VoIP communication using SloMo. We found that downclocked VoIP using SloMo only significantly degrades call quality when network conditions are poor, as expected, but otherwise delivers equivalent Mean Opinion Scores (MOS) for calls. To stress SloMo's downclocking implementation, we also evaluated application throughput at both 1 Mbps and 2 Mbps link rates using `iperf` with 1,000-byte UDP packets. The 1-Mbps results track the packet reception results in Figure 4(a) very closely. SloMo can also take full advantage of 2-Mbps link rates under stable network conditions: application throughputs at 2 Mbps are double those at 1 Mbps. Finally, in addition to evaluating SloMo with the Intel WiFi card, we also performed similar throughput experiments between the Sora node running SloMo and a Macbook Pro laptop with an Apple Airport Extreme WiFi card using the Broadcom BCM43xx firmware. Both downclocked re-

ception and transmission performed as expected between SloMo and the Macbook.

6 Trace-based energy evaluation

Our experiments with the SloMo implementation demonstrate the feasibility and performance of downclocked 802.11 communication. Next we evaluate the potential energy savings when using downclocking in the context of contemporary smartphones and popular apps.

6.1 Methodology

Since we could not directly measure the power consumption of a downclocked WiFi chipset in an actual smartphone, we construct a power model based on measurements of a real device. We also collect MAC-layer packet traces of a variety of real apps running on two different smartphones. We use these packet traces to infer the instantaneous power state of the smartphones' WiFi chipsets and compute the total energy cost for each phone based on the power model.

6.1.1 WiFi power model

Similar to Mittal *et al.* [19], we parameterize our smartphone WiFi power model on the measurements of a Nexus One reported by Manweiler and Choudhury [18]. When actively transmitting and receiving frames, a WiFi chipset must be in a high power state. Once a network transfer completes, the card moves to the idle state. If there is no network activity for a while, the card transitions to the light sleep mode. The light sleep state still consumes a significant amount of power in anticipation of efficiently waking up for incoming traffic. On the Nexus One, the light sleep tail time is roughly 500 ms; if no further network activity occurs, the card returns to the deep sleep state. Table 2 summarizes the model parameters we used. Most are reproduced from [18, 19]; the downclocked values marked with an asterix are estimated as follows.

WiFi power consumption falls into two parts, the analog front-end P_a and the digital processing logic P_d. In the sleep state, the digital logic part is turned off. Given the description of the two sleep modes, we infer that the power difference between them is due to the analog front end remaining functional in light sleep mode but turned off in deep sleep mode. Therefore, we use the light sleep state power as an estimate for the analog power consumption P_a. We then estimate the downclocked power consumption as proportional to the full digital power consumption P_d/α, where α is the clock scaling ratio. When downclocking by a factor of 4, for instance, α at best would be 4 as well. Since it is likely that a practical implementation would experience suboptimal scaling, we conservatively choose $\alpha = 2$ to obtain a lower bound estimate. Note also that the analog part P_a for Tx is greater than Rx since transmission includes an additional power amplifier component. We use the difference between Rx and Tx power (100 mW) from the measurements in previous work to approximate the power consumption of the amplifier.

Parameter	Full Clock / Downclocked (1/4)
Beacon Interval (ms)	100 / 100
Beacon Wakup Period (ms)[4]	2.5 / 2.5
Light Sleep Tail time (ms)	500 / 500
Deep Sleep Power (mW)	10 / 10
Light Sleep Power (mW)	120 / 120
Beacon Wakeup Power (mW)	250 / 185*
Idle Power (mW)	400 / 260*
Rx Power (mW)	600 / 360*
Tx Power (mW)	700 / 460*

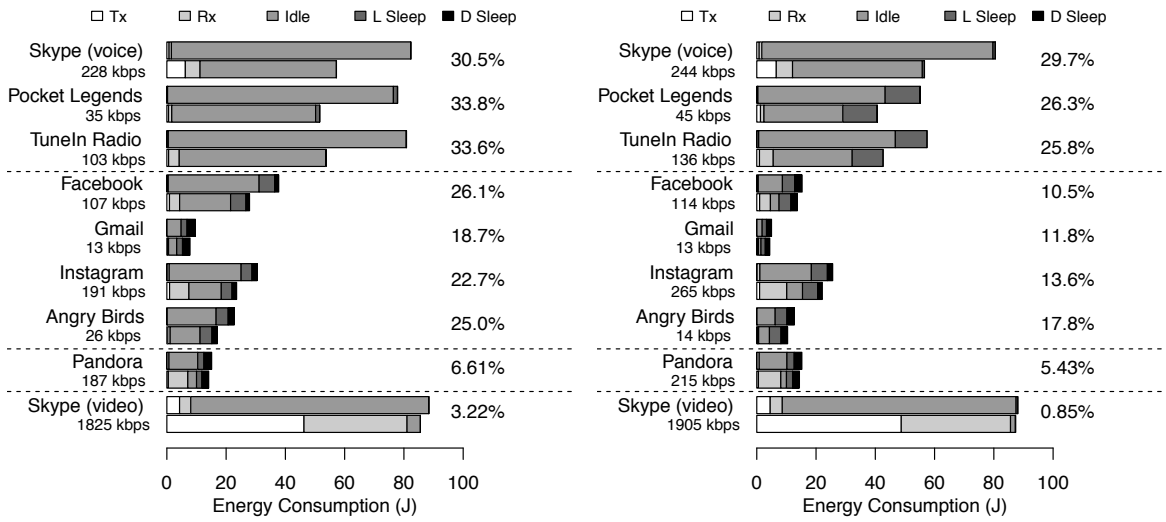
Table 2: WiFi Power Characteristics

6.1.2 Smartphone app traces

To comprehensively evaluate the benefits of SloMo, we sampled a wide range of popular smartphone apps (each has at least 5 million downloads). These nine apps include familiar Internet services like Facebook and Gmail, as well as smartphone-specific services like Pocket Legends (a real-time massively multiplayer game) and TuneIn Radio (a streaming audio service). They differ significantly in the way they interact with the network, spanning interactive real-time traffic to content prefetching to intensive data rates.

We collect high fidelity WiFi packet traces [28] by configuring two MacBook Pro laptops as sniffer nodes in the vicinity of the smartphone and the AP, respectively, and merge the two traces to minimize frame losses. To eliminate bias due to starting and closing the app, we only record a trace when an app is in steady state. Each such capture session lasts for 200 seconds. Finally, to avoid tying our conclusions to a particular smartphone platform, we conduct our experiments on the Google Samsung Nexus S (Nexus) and the Apple iPhone 4S (iPhone). We collected the traces with 4–5 other WiFi devices concurrently using the network, and we emulated a typical SNR scenario where the AP and the wireless station are in the same building but different rooms (i.e., no line-of-sight between the two). Since WiFi devices signal the AP of their intention to sleep and wake up, we are able to faithfully recreate the power state transitions of the WiFi cards on the smartphones using the captured network traces.

[4] We observe the Nexus One employing a variety of beacon wakeup periods (2.5,5,10 ms) on the power measurement trace obtained from the authors of [18]; we use 2.5 ms in our model to be conservative.



(a) Google Nexus S

(b) iPhone 4S

Figure 5: Energy cost of various apps under 802.11 PSM and SloMo. For each app, the upper bar corresponds to the breakdown of energy consumption under 802.11 PSM while the lower bar corresponds to SloMo. The number at the end of the bar group shows the relative energy saving of SloMo over PSM, the higher the better. We also report the bi-directional MAC layer data rate.

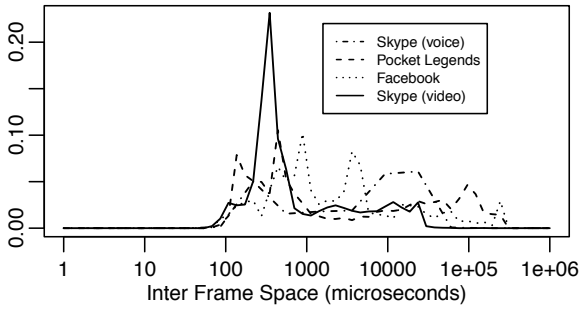


Figure 6: PDFs of the IFT for a selected set of apps on Neuxs S. We remove the inter frame time (SIFS) between DATA and ACK frame for better presentation. IFTs larger than a sleep period are also removed.

6.2 SloMo energy consumption

Figures 5 and 6 combined show the energy and timing behavior of the apps. Figure 5 compares the network energy costs of the apps by power state when using standard 802.11 PSM (top bar) and when using downclocked communication with SloMo (bottom bar). To emphasize energy consumption, we assume SloMo operates at 2 Mbps and the data rates for PSM are the ones reported by the packet capture software. The graphs show results for running the apps on both the Nexus and iPhone. The trends for both phones are similar, but since the iPhone has shorter idle tail times (30–90 ms in our traces versus 220 ms for the Nexus) the benefits of SloMo are smaller for the iPhone than the Nexus.

Figure 5 shows that a wide range of popular apps benefit from SloMo, but they do so for different reasons. To provide insight into the different app behaviors, Figure 6 shows the PDFs of the inter-frame times (IFTs) for four distinctive apps.

Energy consumption in the first group of apps (Skype Voice, Pocket Legends, TuneIn Radio) is dominated by time spent in the idle listening state. Since WiFi cards still consume substantial energy while idle (Table 2), downclocking significantly reduces idle state energy consumption [39]. And since these apps have low data rates, the energy saved during idle listening far exceeds the additional energy consumed for slower data transmission and reception, resulting in energy savings of 30–34% overall on the Nexus. Although these apps have low data rates, their network behavior prevents them from entering sleep mode while idle and makes them relatively power-hungry: As real-time apps, they send and receive packets at frequencies that keep the WiFi card awake in constant active mode (CAM). Figure 6 shows that Skype Voice exchanges packets roughly every 10 ms, and that the Pocket Legend client exchanges game updates with its server as a burst of packets every 100 ms (the peak near 100 μs is the IFT between packets in a burst). TuneIn Radio similarly keeps the WiFi card awake for frequent incoming packets (curve not shown for clarity).

The next group of apps (Facebook, Gmail, Instagram) interact with the network much more intermittently at human time scales. Users navigate through the app and download bursts of content, with pauses in between (e.g.,

Instagram had an average pause time of 1.5 seconds). For such apps, the WiFi card wakes up intermittently when downloading content, and transitions first to idle and then to sleep mode during the longer pause times. Even so, SloMo can still reduce energy consumption during the idle tail time after intermittent network activity and, to a minor effect, during the sleep states. Again, the benefits of downclocking and saving energy during these states outweigh (by 19–26%) the additional energy spent transmitting and receiving at low data rates.

Angry Birds is a good example of many "offline" free apps. Although the game itself does not require network interaction, the embedded ads in the free version cause the app to have similar network characteristics as Facebook and Instagram. The app has intermittent network activity uploading user information and downloading tailored ads, but after each interaction the WiFi card enters the idle state before transitioning to sleep. As a result, Angry Birds spends over 95% of its network energy in the idle tail time, which can account for 65–75% of the entire app energy consumption [21]. (Although the data rate of Angry Birds is just 14% of Instagram, it consumes comparable network energy.) Once again downclocking can substantially reduce energy consumption in the idle state for a 25% savings overall.

Although a music streaming service, Pandora differs from the previous apps in that it prefetches entire songs at a time. In our trace, it downloads a song in the first 10 seconds and has very little network activity for the next 60 seconds. With this behavior, Pandora already uses the network efficiently. Although SloMo does reduce energy consumption by downclocking during the idle and sleep states, it correspondingly increases it for reception and on balance only marginally improves total consumption.

Finally, Skype Video exhibits a similar tradeoff as Pandora. The energy saved by SloMo in downclocking during idle time is matched by the energy expended in using the network at low data rates. In terms of network energy, Skype Video is a wash. As we discuss below, however, SloMo is a poor choice for this kind of app because of the channel airtime it consumes.

6.3 Network impact

Given that SloMo trades off data rates for energy consumption, it is also important to consider the overall network impact due to the use of slower data rates by SloMo in terms of channel airtime. As discussed in Section 3.3, an app might save itself energy by downclocking but unduly impact other devices on the network by consuming more airtime using lower data rates.

Figure 7 shows the channel airtime breakdown of the apps on the Nexus S. It compares the time spent in the states when the apps use standard 802.11 PSM (top bar) and SloMo (bottom bar). The number to the right of each paired bar denotes the contraction in free channel airtime for using SloMo with the app. For instance, the free channel airtime for Skype Voice using PSM divided by the free channel airtime using SloMo is 1.15. The graph shows that downclocking with SloMo does cause the apps to spend more time in actively transmitting and/or receiving. For all apps except Skype Video, the impact on free channel airtime is modest, with contractions ranging between 1.02–1.15. With the much higher data rates of Skype Video, though, using SloMo causes the app to spend most of its time receiving and transmitting data, greatly reducing the free channel airtime compared to PSM. The channel airtime results for the iPhone 4S are very similar (the largest contraction ratio is 1.16 for apps other than Skype Video).

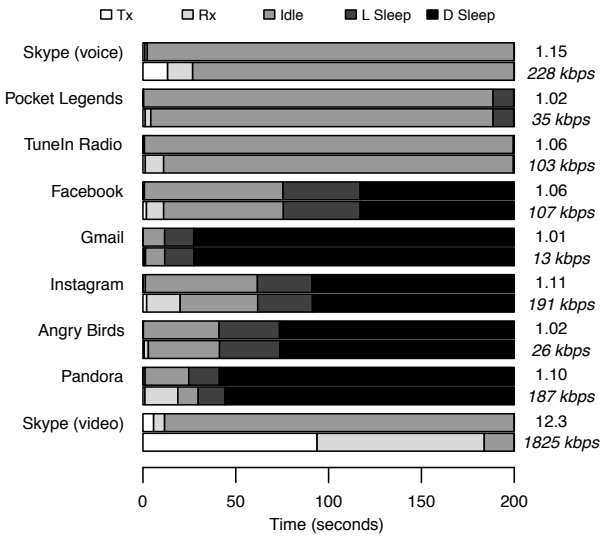


Figure 7: Comparing the timing breakdown for various apps under 802.11 PSM (upper bar) and SloMo on Google Nexus S (lower bar). The number at the end of the bar group shows the free channel airtime contraction ratio, the lower the better with 1.0 as the minimum.

6.4 Alternative approaches

So far we have compared SloMo with current WiFi implementations using PSM. As the traces revealed, though, a critical source of network energy consumption is the tail time of the idle state. Of course, other solutions have been proposed to address this issue as well. As a final evaluation, we compare SloMo with two other approaches, U-APSD [38] and E-MiLi [39], from industry standards and the research community, respectively.

U-APSD. When traffic patterns are periodic, predictable, and symmetric, such as real-time VoIP traffic, the Unscheduled Automatic Power Save Delivery (U-APSD) optimization (defined by the 802.11e standard [38]) could allow devices to enter the sleep state immediately after network activity and avoid the stan-

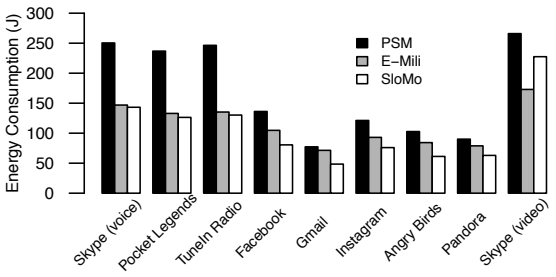


Figure 8: Comparing energy consumption among PSM, E-MiLi and SloMo on the Nexus trace set.

dard tail time in the idle state. Based upon the U-APSD specification, we emulated its use[5] for the Skype Voice and Video apps using the Nexus traces and estimate impressive energy savings of 56% and 44%, respectively, compared with savings of 30.5% and 3.2% using SloMo. Although clearly better in the ideal case of Skype, as noted by others [24] U-APSD is not a general optimization because its effectiveness depends greatly on the degree of symmetry in the traffic. For real-time apps where the traffic pattern is asymmetric, such as Pocket Legends and TuneIn Radio in our examples, U-APSD would not apply. Further, U-APSD is not suitable for intermittent traffic, such as with the Facebook and Gmail apps, which could lead to unnecessary energy waste due to frequent polling of the AP [27].

E-MiLi. E-MiLi redesigns the addressing mechanism of WiFi devices, enabling receivers to determine whether traffic is addressed to them without leaving a low-power listening state [39]. We emulate the use of E-MiLi on our Nexus app traces based upon the WiFi power model and measurements reported by the E-MiLi authors.[6] To facilitate the comparison, we apply the E-MiLi power model to SloMo in contrast to our previous experiments.[7] Figure 8 compares the network energy consumption of PSM, SloMo and E-MiLi on the Nexus apps traces (results were similar for the iPhone traces). Across all apps, downclocking with SloMo saves on average 37.5% energy relative to the default PSM, about 10% more than the 27.7% savings achieved with E-MiLi. For the initial three real-time apps, both SloMo and E-Mili obtain comparable savings. For the others, SloMo performs significantly better than E-Mili, while E-MiLi performs significantly better on Skype Video.

[5] We attempted to purchase U-APSD compliant APs and WiFi cards to experiment with a real implementation, but could not find a hardware, OS, and driver combination that enabled its use in practice.

[6] We measure a WiFi card (Atheros AR9380) from the same manufacturer as the published E-MiLi results to obtain details regarding the power consumption of the sleep state not reported in the E-MiLi paper. The card wakes up at every beacon interval and stays awake for 20 ms before going back to sleep.

[7] As a result, the SloMo energy savings are 10–20% larger relative to PSM compared to the results presented in Figure 5(a).

7 Conclusion

Downclocked 802.11 reception and transmission using compressive sensing is both beneficial and practical. Analysis of the network traffic of a wide range of popular smartphone apps shows that downclocking has the potential to reduce WiFi power consumption on contemporary smartphones by 30%. And our SloMo prototype shows the practicality of implementing downclocking on WiFi clients that communicate seamlessly with unmodified commercial WiFi hardware. While SloMo demonstrates that compressive-sensing techniques are effective for the DSSS encoding used by 802.11b, consumer devices are increasingly employing the higher-rate encodings of 802.11a/g/n. Since the OFDM-based modulation scheme used by these protocols does not share the same inherent spectral sparsity, a tantalizing challenge going forward is to what extent alternative techniques can achieve the same goals for these modulations.

Acknowledgments

We would like to thank Justin Manweiler and Romit Roy Choudhury for providing additional details about their power measurements. We would also like to thank Heather Zheng, our shepherd, and the anonymous reviewers for their detailed feedback. Finally, we are grateful to Lijuan Geng for her assistance with some of the experiments, and Kun Tan for his support and assistance with Sora. This work was supported by generous research, operational and in-kind support from the UCSD Center for Networked Systems (CNS).

References

[1] Y. Agarwal, R. Chandra, A. Wolman, P. Bahl, K. Chin, and R. Gupta. Wireless Wakeups Revisited: Energy Management for VoIP over Wi-Fi Smartphones. In *Proceedings of ACM MobiSys*, June 2007.

[2] M. Anand, E. Nightingale, and J. Flinn. Self-Tuning Wireless Network Power Management. In *Proceedings of ACM MobiCom*, Sept. 2003.

[3] G. Ananthanarayanan and I. Stoica. Blue-Fi: Enhancing Wi-Fi Performance using Bluetooth Signals. In *Proceedings of ACM MobiSys*, June 2009.

[4] Android Code Project: Add WIFI-MODE-FULL-HIGH-PERF (or similar) to official SDK. `http://code.google.com/p/android/issues/detail?id=15549`, Mar. 2011.

[5] Android Developer Reference: WifiManager. `http://developer.android.com/reference/android/net/wifi/WifiManager.html`.

[6] R. Chandra, R. Mahajan, T. Moscibroda, R. Raghavendra, and P. Bahl. A Case for Adapting Channel Width in Wireless Networks. In *Proceedings of ACM SIGCOMM*, Aug. 2008.

[7] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris. Span: An Energy-Efficient Coordination Algorithm for Topology Maintenance in Ad Hoc Wireless Networks. *Wireless Networks*, 8(5), Sept. 2002.

[8] Y.-C. Cheng, M. Afanasyev, P. Verkaik, P. Benkö, J. Chiang, A. C. Snoeren, S. Savage, and G. M. Voelker. Automating Cross-Layer Diagnosis of Enterprise Wireless Networks. In *Proceedings of ACM SIGCOMM*, Aug. 2007.

[9] CNET. Raziko for Android: Full specs. `http://download.cnet.com/Raziko-for-Android/3000-2168_4-12094384.html`.

[10] M. Davenport, P. Boufounos, M. Wakin, and R. Baraniuk. Signal Processing With Compressive Measurements. *IEEE Journal of Selected Topics in Signal Processing*, 4(2):445–460, Apr. 2010.

[11] W. R. Dieter, S. Datta, and W. K. Kai. Power Reduction by Varying Sampling Rate. In *Proceedings of the IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, pages 227–232, Aug. 2005.

[12] F. R. Dogar, P. Steenkiste, and K. Papagiannaki. Catnap: Exploiting High Bandwidth Wireless Interfaces to Save Energy for Mobile Devices. In *Proceedings of the ACM MobiSys*, June 2010.

[13] FitBit Aria: Technical Specs. `http://www.fitbit.com/product/aria/specs`.

[14] GE Healthcare Dash 5000 Patient Monitors. `http://www3.gehealthcare.com/en/Products/Categories/Patient_Monitoring/Patient_Monitors/Dash_5000`, Nov. 2011.

[15] D. Halperin, B. Greenstein, A. Seth, and D. Wetherall. Demystifying 802.11n Power Consumption. In *Proceedings of USENIX HotPower*, Oct. 2010.

[16] R. Krashinsky and H. Balakrishnan. Minimizing Energy for Wireless Web Access with Bounded Slow Down. In *Proceedings of ACM MobiCom*, Sept. 2002.

[17] J. Liu and L. Zhong. Micro Power Management of Active 802.11 Interfances. In *Proceedings of ACM MobiSys*, June 2008.

[18] J. Manweiler and R. R. Choudhury. Avoiding the Rush Hours: WiFi Energy Management via Traffic Isolation. In *Proceedings of ACM MobiSys*, June 2011.

[19] R. Mittal, A. Kansal, and R. Chandra. Empowering Developers to Estimate App Energy Consumption. In *Proceedings of ACM MobiCom*, Aug. 2012.

[20] A. Nicholson and B. Noble. Breadcrumbs: Forecasting Mobile Connectivity. In *Proceedings of ACM MobiCom*, Sept. 2008.

[21] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app? Fine Grained Energy Accounting on Smartphones with Eprof. In *Proceedings of ACM EuroSys*, Apr. 2012.

[22] F. I. D. Piazza, S. Mangione, and I. Tinnirello. On the Effects of Transmit Power Control on the Energy Consumption of WiFi Network Cards. In *Proceedings of QSHINE*, Nov. 2009.

[23] J. G. Proakis and M. Salehi. *Digital Communications*. McGraw-Hill, Nov. 2007.

[24] A. J. Pyles, Z. Ren, G. Zhou, and X. Liu. SiFi: Exploiting VoIP Silence for WiFi Energy Savings in Smart Phones. In *Proceedings of ACM UbiComp*, Sept. 2011.

[25] J. N. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits*. Prentice Hall, 2nd edition, Jan. 2003.

[26] A. Rahmati and L. Zhong. Context-for-Wireless: Context-Sensitive Energy-Efficient Wireless Data Transfer. In *Proceedings of ACM MobiSys*, June 2007.

[27] E. Rozner, V. Navda, R. Ramjee, and S. Rayanchu. NAPman: Network-Assisted Power Management for WiFi Devices. In *Proceedings of ACM MobiSys*, June 2010.

[28] A. Schulman, D. Levin, and N. Spring. On the Fidelity of 802.11 Packet Traces. In *Proceedings of the 9th PAM Conference*, Apr. 2008.

[29] L. Shang, L.-S. Peh, and N. K. Jha. Dynamic Voltage Scaling with Links for Power Optimization of Interconnection Networks. In *Proceedings of IEEE HPCA*, Jan. 2003.

[30] C. E. Shannon. Communication in the Presence of Noise. *Proceedings of the Institute of Radio Engineers*, 37(1):10–21, Jan. 1949.

[31] E. Shih, P. Bahl, and M. Sinclair. Wake on Wireless: An Event Driven Energy Saving Strategy for Battery Operated Devices. In *Proceedings of ACM MobiCom*, Sept. 2002.

[32] K. Tan, J. Zhang, H. Wu, F. Ji, H. Liu, Y. Ye, S. Wang, Y. Zhang, W. Wang, and G. M. Voelker. Sora: High Performance Software Radio Using General Purpose Multicore Processors. In *Proceedings of NSDI*, Apr. 2009.

[33] J. Tropp, J. Laska, M. Duarte, J. Romberg, and R. Baraniuk. Beyond Nyquist: Efficient Sampling of Sparse Bandlimited Signals. *IEEE Transactions on Information Theory*, 56(1):520–544, Jan. 2010.

[34] P. Verkaik, Y. Agarwal, R. Gupta, and A. C. Snoeren. Softspeak: Making VoIP Play Well in Existing 802.11 Deployments. In *Proceedings of NSDI*, Apr. 2009.

[35] Google Play Store: VirtualRadio — What's New. `http://bit.ly/SVgzMG`.

[36] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. In *Proceedings of OSDI*, Nov. 1994.

[37] Wi-Fi HVAC Monitor. `http://bit.ly/V6JNpn`.

[38] Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications amendment 8: Medium Access Control (MAC) Quality of Service Enhancements, IEEE 802.11e, 2005.

[39] X. Zhang and K. G. Shin. E-MiLi: Energy-Minimizing Idle Listening in Wireless Networks. In *Proceedings of ACM MobiCom*, Sept. 2011.

Splash: Fast Data Dissemination with Constructive Interference in Wireless Sensor Networks

Manjunath Doddavenkatappa, Mun Choon Chan, Ben Leong
National University of Singapore

Abstract

It is well-known that the time taken for disseminating a large data object over a wireless sensor network is dominated by the overhead of resolving the contention for the underlying wireless channel. In this paper, we propose a new dissemination protocol called *Splash*, that eliminates the need for contention resolution by exploiting constructive interference and channel diversity to effectively create fast and parallel pipelines over multiple paths that cover all the nodes in a network. We call this *tree pipelining*. In order to ensure high reliability, Splash also incorporates several techniques, including exploiting transmission density diversity, opportunistic overhearing, channel-cycling and XOR coding. Our evaluation results on two large-scale testbeds show that Splash is more than an order of magnitude faster than state-of-the-art dissemination protocols and achieves a reduction in data dissemination time by a factor of more than 20 compared to DelugeT2.

1 Introduction

A data dissemination protocol, like Deluge [14], is a fundamental service required for the deployment and maintenance of practical wireless sensor networks because of the need to periodically re-program sensor nodes in the field. Existing data dissemination protocols employ either a contention based MAC protocol like CSMA/CA [6, 5, 7, 10, 12, 30, 18, 14] or TDMA [17] for resolving the multiple access problem of the wireless channel. As there is a large amount of data that needs to be disseminated to all the nodes in the network, there is often severe contention among the many transmissions from many nodes. Existing MAC protocols incur significant overhead in contention resolution, and it has been shown that Deluge can take as long as an hour to program a 100-node sensor network [27].

In this paper, we propose a new data dissemination protocol, called *Splash*, that completely eliminates contention overhead by exploiting constructive interference. Splash is scalable to large, multi-hop sensor networks and it is built upon two recent works: Glossy [9] and PIP [24]. Glossy uses constructive interference in practical sensor networks to enable multiple senders to transmit the same packet simultaneously, while still allowing multiple receivers to correctly decode the transmitted packet. Like Glossy, we eliminate the overhead incurred in contention resolution by exploiting constructive interference. Raman et al. showed in PIP that a pipelined transmission scheme exploiting channel diversity can avoid self interference and maximize channel utilization for a single flow over multiple hops by ensuring that each intermediate node is either transmitting or receiving at any point of time. Splash uses constructive interference to extend this approach to *tree pipelining*, where each level of a dissemination tree serves as a stage of the pipeline.

While the naive combination of synchronized and pipelined transmissions achieves substantial gains in the data dissemination rate by maximizing the transmission opportunities of the senders, it also creates a significant reliability issue at the receivers. First, in order to improve efficiency, we need to use a large packet size (i.e. at least 64 bytes). However, increasing packet size reduces the reliability of constructive interference as the number of symbols to be decoded correctly increases [9]. Second, channel quality varies significantly among different channels, and there are typically only a small number of available channels that are of sufficiently good quality. If a poor channel is chosen for a stage of the pipeline, the pipeline transmission may be stalled.

Splash includes a number of techniques to improve the packet reception rate. (1) We improve the reception rates over all receivers by exploiting transmitter density diversity by varying the number of transmitters between transmission rounds. When the sets of transmitters are varied, the sets of receivers that can decode the synchronized transmissions correctly also change. Hence, different sets of nodes are likely to correctly decode packets

during different transmission rounds. The challenge is to maximize the differences among different transmission rounds. (2) We increase reception opportunities by incorporating opportunistic overhearing which involves early error detection and channel switching. A node in Splash identifies a corrupted packet on-the-fly during its reception and switches its channel to overhear the same packet when it is being forwarded by its peer nodes in the dissemination tree. (3) We exploit channel diversity to improve packet reception ratio by varying the channels used between different transmission rounds. This is particularly important since the use of the same bad channel can stall the pipeline transmission consistently. (4) Finally, we utilize a simple XOR coding scheme to improve packet recovery by exploiting the fact that most receivers would have already received most of the packets after two transmission rounds.

We implemented Splash in Contiki-2.5 and we evaluated the protocol on the Indriya testbed [3] with 139 nodes and the Twist testbed [13] with 90 nodes. We compare Splash to both Deluge [14] in Contiki and to the much improved DelugeT2 implemented in TinyOS. As we use DelugeT2 as a baseline, it allows us to compare Splash to many of the existing dissemination protocols in the literature as most of them are also compared to Deluge. Our results show that Splash is able to disseminate a 32-kilobyte data object in about 25 seconds on both the testbeds. Compared to DelugeT2, Splash reduces dissemination time on average by a factor of 21, and in the best case, by up to a factor of 57.8. This is significantly better than MT-Deluge [10], the best state-of-the-art dissemination protocol, which achieves a reduction factor of only 2.42 compared to Deluge.

The dissemination performance of our current implementation of Splash achieves a *network-wide* goodput of 10.1 kilobits/sec per node for a multihop network of 139 nodes with up to 9 hops. Splash's goodput is higher than that of all the network-wide data dissemination protocols [6, 5, 7, 10, 12, 30, 18, 14, 17] previously proposed in the literature. Splash's performance is comparable to Burst Forwarding [8], the state-of-the-art pipelined bulk transfer protocol over TCP for sensor networks, which is able to achieve a goodput of up to 16 kilobits/sec, but only for a single flow over a single multihop path.

Finally, Splash is also significantly more compact than DelugeT2 in terms of memory usage. Splash uses 9.63 and 0.68 kilobytes less ROM and RAM respectively than DelugeT2. Given that it is not uncommon for sensor devices to have only about 48 and 10 kilobytes of ROM and RAM respectively, these are significant savings in memory, that will be available for use by sensor applications.

The rest of the paper is organized as follows. In Section 2, we discuss the related work. Section 3 presents our measurement study of constructive interference on a practical testbed. We present Splash and the details of its implementation in Section 4. Section 5 presents our evaluation results on the Indriya and Twist testbeds. Finally, we conclude in Section 6.

2 Related Work

In their seminal work on Glossy [9], Ferrari et al. showed that constructive interference is practical in wireless sensor networks. They observed that there is a high probability that the concurrent transmissions of a same packet will result in constructive interference if the temporal displacement among these transmissions is smaller than 0.5 microsecond. The implementation of Glossy is able to meet this requirement and a small packet can be flooded to all nodes with deterministic delays at the relay nodes which allows accurate network-wide synchronization. Glossy is designed to flood a single packet at a time, e.g., a control packet. On the other hand, a dissemination protocol needs to achieve bulk transfer of large packets, which introduces a new set of problems such as the need for 100% reliability, pipelining, channel switching, and scalability in terms of both network size and constructive interference.

The scalability of constructive interference was recently studied by Wang et al. [28]. They showed that the reliability of constructive interference decreases significantly when the number of concurrent transmitters increases, where *reliability* is defined as the probability that a packet that is concurrently transmitted by multiple transmitters will be decoded correctly at a receiver. While [28] is the first work to study this problem, it is based on theory and simulations, and does not include any experimental evaluation. Our empirical results show that the scalability problem highlighted is actually more severe in practice. Wang et al. also proposed Spine Constructive Interference based Flooding (SCIF) to mitigate the scalability problem, but the correctness of SCIF assumes many conditions that are hard to achieve in practice. For example, length of a network cell is half of the radio communication range. In contrast, our strategy for handling the scalability problem is a fully practical solution based on collection tree protocols such as CTP [11] and the observation that typically more than 50% of nodes in a collection tree are leaf nodes even at the lowest transmission power where the underlying network is connected [4].

A key challenge in implementing pipelining over a multihop path is self interference: a node's next packet can interfere with its immediate previously forwarded packet. There are two common solutions. First, we can introduce inter-packet gaps such that the previous packet would be out of the interference range before attempting to transmit the next packet [15]. However, this method

would drastically reduce the end-to-end throughput as a long gap of 5 packet transmission times is required for a single flow in practice [15]. Moreover, in the case where multiple data flows are active, this method is ineffective because of inter-flow interference. The second solution is to exploit channel diversity [23, 24, 8]. However, we observe that this approach ignores two practical issues that can severely degrade the performance of its packet pipeline. First, although the IEEE 802.15.4 standard defines 16 non-overlapping channels, the number of channels of usable quality is typically much smaller in practice because of various factors, e.g., interference from WiFi channels [21]. Second, the approach ignores the fact that links for routing are typically chosen on the best available channel, and the performance of other channels on such links can be poor in practice. These two issues can severely degrade the performance by stalling the packet pipeline.

As dissemination is a fundamental service in sensor networks, there are numerous protocols in the literature [6, 7, 10, 12, 18, 14, 17]. Typically, they are epidemic approaches incorporating special techniques in order to reduce the incurred overhead. Such techniques include Trickle suppression [20], network coding [12], exploiting link qualities [6], virtual machines [19], etc. While existing protocols differ in their techniques, they all share a common feature that they employ a MAC protocol like CSMA/CA or TDMA for contention resolution, and typically their dissemination times are in the order of minutes for disseminating full images in practical networks. Our goal in this paper is to completely eliminate contention overhead by exploiting constructive interference and we show that by doing so, we can reduce the dissemination time by an order of magnitude compared to existing approaches.

3 Measurement Study

To understand the behavior of simultaneous transmissions in real-world setups, we conducted a measurement study of constructive interference on the Indriya [3] wireless sensor testbed. In particular, we studied the scalability of simultaneous transmissions and correlation among packet receptions across different nodes decoding such transmissions.

We used the code from the Glossy [9] project in our experiments, our experimental methodology is similar to that adopted by Ferrari et al. in [9]. An initiator node broadcasts a packet to a set of nodes which in turn forward the received packet concurrently back to the initiator. This results in constructive interference at the initiator, where we measured the reliability of the reception. Since our goal is to use constructive interference for the dissemination of large objects, we used the maximum packet size of 128 bytes in our experiments. In addition, the payload of each packet was randomized. Our experiments were carried out on the default Channel 26, unless specified otherwise. Channel 26 is one of the only two ZigBee channels that does not overlap with the commonly used WiFi channels [21].

3.1 Scalability

In Fig. 1, we plot the reliability of packet reception against the number of concurrent transmitters for three randomly chosen initiators on three different floors of the Indriya testbed. In each experiment, both the initiator and the randomly chosen set of concurrent transmitters were located on the same floor. We recorded over 1,000 packet transmissions on each floor on Channel 26. We see from Figs. 1(a) and 1(b) that reliability generally decreases when there are more concurrent transmitters.

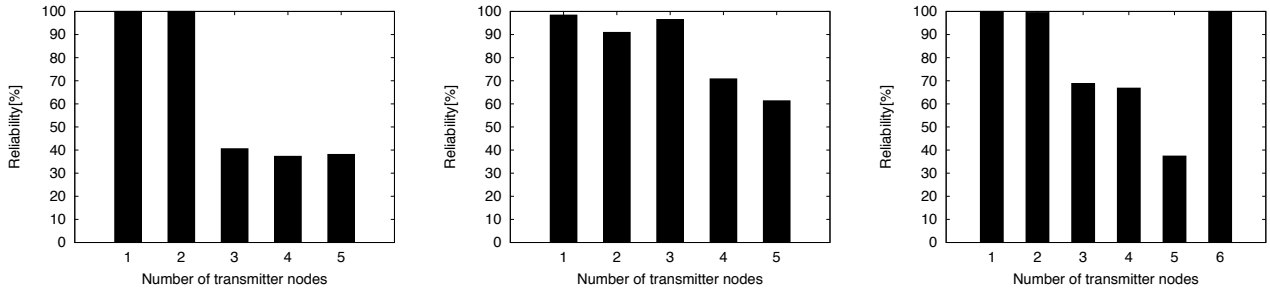
In fact, it had been shown by Wang et al. [28] through analytical model and simulation that the reliability of constructive interference decreases when the number of concurrent transmitters increases, due to the increase in the probability of the maximum time displacement across different transmitters exceeding the required threshold for constructive interference. Our measurements suggest that the highlighted problem is more severe in practice, and even a small number of three to five concurrent transmitters can significantly degrade the reception at a receiver.

However, it is sometimes possible for an increase in the number of concurrent transmitters to result in improved reception reliability. In particular, we see in Fig. 1(c) that by adding a sixth node, the reliability increases from about 37% to 100%. This is likely caused by the *capture effect* since the sixth node was located some 2 meters away from and within line of sight of the initiator.

This suggests that the impact of the number of transmitters (transmission density) on reception reliability does not follow a fixed trend like what was predicted by Wang et al. [28]. But depends also on the positions of the concurrent transmitters relative to the receiver. So, instead of attempting to determine the optimal transmission density, we can try to transmit at both high and low transmission densities to improve reception reliability.

3.2 Receiver Correlation

In existing dissemination protocols, it is common for a node to attempt to recover missing packets from its neighbors. It is hence important for us to understand the correlation of the packets received by neighboring receivers. While Srinivasan et al. had previously investigated the correlation of packets received by the receivers



(a) Floor 1 (b) Floor 2 (c) Floor 3

Figure 1: Plot of reliability against the number of concurrent senders.

in a sensor network [26], they did not study the correlation in the presence of constructive interference.

To this end, we set up an experiment involving 21 nodes spanning an area of 30m $\times$ 30m on the 3rd floor of Indriya. One node was designated as the initiator node, ten nodes were randomly chosen to serve as relays, and the remaining ten were used as receivers. The initiator broadcasts a packet once every second over a duration of four hours and the relay nodes forward the packet concurrently, which results in constructive interference at the various receiver nodes. As Srinivasan et al. had earlier shown that WiFi interference is the most likely reason for correlations in packet reception [26], we repeated this experiment on two separate channels: Channel 26, which is non-overlapping with the WiFi channels occupied in the building where Indriya is deployed, and Channel 22, which overlaps with an occupied WiFi channel.

We investigated the correlation among the packet receptions at the receiver nodes (R) by computing the Pearson's correlation coefficient at a granularity of one packet. We present the coefficient values for Channels 26 and 22 in Table 1. Note that as a coefficient matrix corresponding to a channel is symmetric, we represent data corresponding to the two channels in a single table (matrix). The values in the lower half of the table (below the diagonal) correspond to Channel 26 and the upper half corresponds to Channel 22.

As expected, for Channel 26, which does not overlap with an occupied WiFi channel, the correlation coefficients are small. This suggests that the packet receptions across different receivers are effectively independent. On the other hand, for Channel 22, which overlaps with an occupied WiFi channel, the coefficients are relatively large, indicating that there is significant correlation in the reception at the various receivers. Our results suggest that it might be hard for a node to recover missing packets from its neighbors if a noisy channel like Channel 22 is used, since many neighboring nodes would likely be missing the same packets.

Table 1: Correlation coefficients observed on Channel 26 (lower half) and Channel 22 (upper half).

R	1	2	3	4	5	6	7	8	9	10
1	**1.0**	.56	.62	.64	.57	.58	.60	.52	.55	.58
2	.04	**1.0**	.52	.63	.51	.54	.46	.53	.50	.55
3	0.0	-.02	**1.0**	.55	.48	.56	.46	.44	.46	.49
4	.05	.23	0.0	**1.0**	.61	.61	.52	.63	.59	.68
5	.04	.07	.01	.13	**1.0**	.51	.52	.51	.61	.53
6	.03	.09	-.01	.13	.03	**1.0**	.46	.48	.50	.53
7	.03	.12	0.0	.16	.06	.09	**1.0**	.45	.49	.47
8	.02	.11	-.01	.17	.06	.11	.13	**1.0**	.49	.66
9	.02	.03	.01	.06	.08	.02	.05	.02	**1.0**	.49
10	.02	.10	0.0	.15	.10	.09	.17	.21	.05	**1.0**

4 Splash

In this section, we describe Splash, a new data dissemination protocol for large data objects in large sensor networks that completely eliminates contention overhead by exploiting constructive interference and pipelining.

Raman et al. proposed PIP (Packets in Pipeline) [24] for transferring bulk data in a pipelined fashion over a single path of nodes over multiple channels. They exploit channel diversity to avoid self interference by having each intermediate node use a different channel to receive packets. A key insight of this pipeline approach is that at any point in time, an intermediate node is either transmitting or receiving packets and this achieves the maximal utilization of air time.

Splash can be considered as an extension of PIP's approach that incorporates three key innovations to support data dissemination to multiple receivers over multiple paths:

1. *Tree pipelining* which exploits constructive interference to effectively create parallel pipelines over multiple paths that cover all the nodes in a network. In our approach, a collection tree is used in the reverse direction for dissemination which in turn allows us to mitigate the scalability problem of the

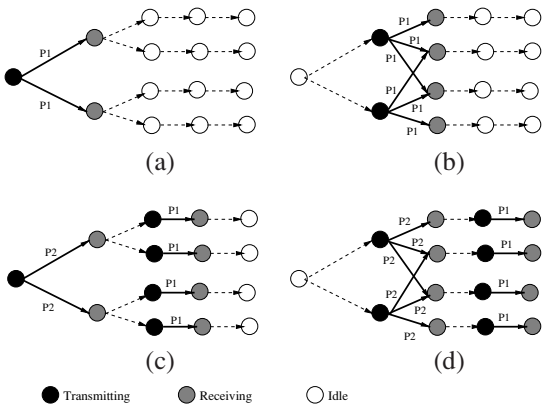


Figure 2: Illustration of pipelining over a tree.

constructive interference and to minimize the differences that exist among the performance of different channels.

2. *Opportunistic overhearing* from peers by exploiting multiple pipelines, which provides each node with more chances of receiving a packet.

3. *Channel cycling* that increases the chance of reusing a good channel while avoiding interference. Different channels are used at different stages of the pipeline between different transmission rounds to avoid stalling of the pipeline in case a bad channel is inadvertently chosen.

In the rest of this section, we discuss in detail various components of Splash and some of its implementation details.

4.1 Tree Pipelining

Splash is the first protocol to exploit constructive interference to support pipelining over a dissemination tree in which each level of the tree acts as one stage of the pipeline. This is illustrated in Fig. 2.

In the first cycle (see Fig. 2(a)), the root node (level zero) transmits the first packet P1. The receivers at the first level, which are synchronized upon receiving P1, will simultaneously forward P1 in the second cycle so that these simultaneous transmissions interfere constructively at the nodes on the second level (see Fig. 2(b)). In the third cycle (see Fig. 2(c)), while nodes at the second level forward P1 to the third level, the root node simultaneously transmits the second packet P2. Note that these simultaneous transmissions of different packets do not interfere with each other as each level of the tree is configured to transmit/receive packets on a different channel. In Fig. 2(c), P2 is transmitted on the receiving channel of the first-level nodes while P1 is transmitted on a different receiving channel for the third-level nodes. Note also that a third-level node will receive transmissions from several second-level nodes, instead of just one. We have omitted some of the transmission arrows in Fig. 2(c) to reduce clutter.

This results in a tree-based pipeline in which packets are disseminated in a ripple-wave-like fashion from the root. Except for the root node (which only transmits), all the nodes are either transmitting or receiving at all times once the pipeline is filled (see Fig. 2(d)). This allows Splash to achieve maximum possible end-to-end throughput.

The tree structure is needed to allow Splash to coordinate transmissions and channel assignment, also to ensure that each transmission is forwarded to every node in the network. Splash uses an underlying collection protocol like CTP [11] to derive its tree structure. We believe that our approach would incur minimal overhead as a CTP-like collection protocol is an integral part of most sensor network applications and we can make use of its existing periodic beacons in order to build the dissemination tree. Moreover, as CTP-like protocols are typically data-driven and they are designed to build stable trees by preferring stability over adaptability [1], diverting some of its periodic beacons for another use will not affect the stability of its data collection tree.

In practice, collection protocols often attempt to use the best links on the best channel (typically Channel 26) to build a tree. However, the performance of the other channels on such links is often not comparable to that of the best channel. So, if a dissemination tree is built using the default channel, the link quality on the same transmitter-receiver pair may be good on the default channel but poor on a different channel. On the other hand, building the dissemination tree on the poorest channel is also not a viable option since the network may not even be connected on such channels. Our approach therefore is to use the best channel (Channel 26) to build the dissemination tree at a lower transmission power but to use the maximum transmission power during dissemination. Our hypothesis is that the performance of different channels at the maximum transmission power is likely be comparable to that of the best channel at a lower transmission power.

Opportunistic Overhearing. In the transmission pipeline, each node is either receiving or transmitting. When a node is unable to successfully decode a transmission, it will be unable to relay the packet to the next stage. In such instances, instead of idling, such a node can switch to listening mode and attempt to recover the missing packet by overhearing the transmissions of its peers on the same level of the dissemination tree. This means that each node effectively has two opportunities

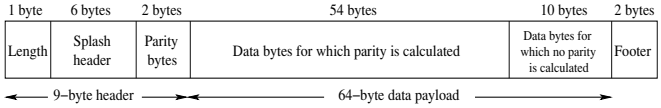


Figure 3: Packet format used in Splash.

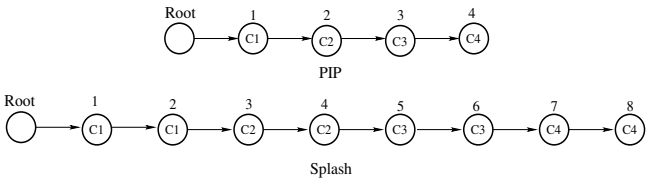


Figure 4: Channel assignment.

to receive a given packet.

The decision to overhear transmissions has to be made before a node has completely received and decoded a packet, because to achieve constructive interference, a node needs to start calibrating its radio for transmission even before the packet to be transmitted is completely read from the radio hardware buffer. By the time a node completely reads, decodes and identifies packet corruption, its peers would have started calibrating their radio for transmission, and they begin transmissions before the node can switch over to overhearing mode which involves calibrating the radio for reception.

In order to address this issue, we add two bytes of parity information of the data payload bytes that are located before the last 12 bytes of the packet as the time required to receive these 12 bytes is the minimum amount of time necessary for verifying packet corruption and to either switch channel for overhearing in the case of corruption or to calibrate the radio for synchronous transmissions otherwise. Fig. 3 depicts format of a Splash packet with its default data payload size of 64 bytes. The parity of the first 54 bytes of data is computed and inserted in the header. This allows a receiving node to detect any corruption in these bytes as soon as it receives the 54^{th} data byte. If bit corruption is detected by the parity check, the reception of the current packet is aborted and the node immediately switches its channel to the receiving channel of its next hop nodes so that it can attempt to overhear the same packet while it is being forwarded by its peers in the next cycle. If corruption occurs within the last 12 bytes of the packet, the packet will not be recoverable with opportunistic overhearing.

4.2 Channel Cycling & Channel Assignment

Channel Cycling. It is well-known that the quality of channels is a function of both temporal and spatial variations. To ensure that nodes do not keep using the same (poor) channel, we use a different channel assignment between different rounds of dissemination in order to reduce the impact of the bad channels. In the case where the root transmits the same packet twice, by incorporating opportunistic overhearing and channel cycling, a node can potentially receive a packet 4 times, and possibly over 4 different channels. If the reception on one of the channels is bad, the packet could possibly be successfully decoded on one of the remaining channels.

We coordinate channel switching between different dissemination rounds of Splash by transmitting a small 7-byte control packet. After every round of dissemination, the control packet is flooded from the root node over the tree pipeline by exploiting constructive interference 20 times. We do so because while there is a probability of some nodes not receiving this packet if we flood it only once, it has been shown that the probability that a node will receive such a small control packet over constructive interference is more than 0.999999 for ten retransmissions on Channel 26 [9]. We flood 20 times for good measure because we do not always use a channel that is as good as Channel 26. Also, we can afford to do so because flooding the packet 20 times takes only a few tens of milliseconds. After the completion of these 20 floods, a node that received the control packet at least once will switch to a pre-assigned channel on which it is expected to receive data packets in the next dissemination round. If a node still fails to receive the control packet, a timeout is used and the node recovers any missing data packets during local recovery.

Channel Assignment. In Fig. 4, we illustrate the channel assignment strategies for PIP and Splash using only four channels (C1, C2, C3, and C4). There are two key advantages of our assignment strategy. First, it allows more efficient channel cycling than PIP's method by allowing to cycle good channels in pairs on consecutive pipeline stages. Second, it supports a longer pipeline if interference extends to several hops as observed in a deployment on the Golden Gate Bridge [15]. However, in our strategy, we need to ensure that we do not use pairs of adjacent channels on consecutive pairs of stages as adjacent channels interfere with each other [29].

In our current implementation of Splash, we choose the ZigBee channels in such a way that they are either non-overlapping or only partially overlapping with the 3 most commonly used WiFi channels (channels 1, 6 and 11). On the testbeds which have network diameters not more than 9 hops, we observed that Splash's channel assignment strategy needs only four such ZigBee channels to avoid any interference.

4.3 Exploiting Transmission Density Diversity

We had shown in Section 3.1 that the effect of the number of transmitters (transmission density) on reception reliability for constructive interference does not follow a fixed trend but depends on the positions of the concurrent transmitters relative to the receiver.

Our key insight is that we can exploit *diversity in transmission density* to improve reliability, not by attempting to determine the optimal number of transmitters, but by transmitting the full data object twice using different transmission densities. In the first round, data is disseminated over the dissemination tree but only non-leaf nodes are asked to transmit. Since typically more than 50% of nodes in a tree are leaf nodes even at the lowest transmission power where the underlying network is connected [4], the number of concurrent transmitters is significantly reduced. In the second round, transmissions are made by all the nodes at each level of the tree. By using more transmitters, some nodes which were not reachable in the first round might now be reached. Moreover, a higher node density is also helpful in specific cases because of the capture effect as we discussed in Section 3.1.

4.4 XOR Coding

After two rounds of dissemination using different transmission densities, we observed in our experiments (see Section 5) that a considerable percentage of the nodes (about 50%) received most but not all the disseminated packets. This is a bad situation for local recovery because even though the number of missing packets may be small, there would be significant wireless contention if too many nodes attempted to recover the missing packets locally from their neighbors. This would significantly reduce the gain achieved through constructive interference by the first two rounds of dissemination.

While it is possible to perform a few more rounds of simple dissemination, we found that the potential gain was limited. This is because the missing packets are different among the different nodes and the root has no way of efficiently determining which exact packets are missing. If all packets are disseminated again, the overhead is very high with minimal gain.

This motivated us to use a third round of dissemination based on XOR coding instead. XOR coding is best suited for recovering missing packets if a node already has most of the packets and only a small portion is missing. Assume that a node already has a fraction p of the total packets. If the degree of the XOR packet is n (i.e. the coded packet is constructed by performing an XOR operation on n packets), then the likelihood that the packet is useful (i.e. that the receiving node had earlier received $n-1$ out of the n packets successfully) is $n(1-p)p^{n-1}$. This likelihood is maximized when $n = \frac{-1}{\ln(p)}$. We found in our experiments that p is about 95% after the first two rounds of dissemination, so in our current implementation, we set $n = 20 \approx \frac{-1}{\ln(0.95)}$.

In the third round, the payload in each packet is the result of 20 randomly chosen packets XORed together. To minimize the overhead, we do not indicate the identities of the packets used in the XOR operations in the packet header. Instead, we use the sequence number of the packet as a seed for choosing these packets based on a predefined pseudo-random function. This allows a receiver to decode packets without any additional overhead. In addition, like the first round of dissemination, only non-leaf nodes participate in forwarding XORed packets in the third round.

Naively, it might seem like it is sufficient to send $\frac{1}{20} = 5\%$ of the total number of packets. However, we found empirically (see Section 5.2) that such an approach is not sufficient to achieve a high packet recovery rate. Instead we send all the original packets with each original packet XORed with 19 randomly chosen packets. This ensures that every single packet is retransmitted at least once, and it also means that the third dissemination round is equivalent to the first two rounds in length.

We also considered using a fountain or rateless code during the "regular" dissemination rounds instead of introducing a third round of simple XOR-coded dissemination. However, we decided not to do so because of the associated decoding costs. In the experiments with Rateless Deluge [12], the decoding process can easily take more than 100 seconds for a 32-kilobyte data object. In comparison, Splash can disseminate the same object in about 25 seconds with simple XOR coding.

4.5 Local Recovery

After three rounds of dissemination, typically about 90% of the nodes would have downloaded the complete data object and most of the remaining nodes would have downloaded most of the object. This makes local recovery practical. Local recovery also allows the nodes to exploit spatial diversity and non-interfering nodes in different parts of the network can simultaneously recover the missing packets from their neighbors.

We implement a very simple CSMA/CA-based local recovery scheme on the default Channel 26. As Splash uses an underlying collection tree protocol to build its dissemination tree, a node will have link quality estimates for its neighboring nodes. A node with missing packets will send a bit vector containing information on the missing packets to a neighbor, starting with the one with the best quality link. If this neighbor has any of the missing packets, it will forward these packets to the

requesting node; if not, the requesting node will ask the next neighbor. If a node reaches the end of its neighbor list and it still has missing packets, it will start querying its neighbors afresh. Because the network is fully connected, this local recovery procedure is guaranteed to converge. Also, as most (about 90%) nodes already have the full data object, it converges quickly (see Section 5.2).

4.6 Implementation Challenges

The key requirement for constructive interference is that nodes have to transmit the same packet at the same time. Glossy satisfies this requirement as a set of nodes receiving a packet are synchronized to the SFD (Start Frame Delimiter) interrupt from the radio hardware (ChipCon2420 (CC2420)) signalling the end of the reception of a packet. Splash is built upon the source code for Glossy [9]. The challenge is to transform the Glossy code into a dissemination protocol while retaining its capability to perform synchronized transmissions.

Channel Switching. First, we added the capability for switching channels for the pipelining operations. Upon receiving a packet, a node switches its channel to that of its next hop nodes, transmits the received packet, and then switch back to its receiving channel to listen for the next packet. Channel switching for transmission has to be performed only after completely receiving an incoming packet and before submitting the transmit request to the radio for forwarding the received packet. The time taken for channel switching cannot vary too much across nodes as such variations desynchronize their submission of the transmit request.

On the other hand, as the clocks of microcontrollers are not synchronized across nodes, the time taken for channel switching can vary from node to node. Our goal is to minimize such variations by enabling channel switching by executing only a minimal number of instructions between the completion of the reception of a packet and the submission of the request for its transmission (forwarding).

The operation of channel switching involves writing to the frequency control register of the radio hardware and then calibrating the radio for transmission. The action of writing to a register in turn involves enabling the SPI (Serial Peripheral Interface) communication by pulling down a pin on the radio, communicating the address of the register to be written, writing into the register and finally disabling the SPI access. Similarly, radio calibration involves enabling the SPI, transmitting a command strobe requesting for calibration and disabling the SPI. While the actual operations of calibration and register access take more or less constant time, enabling the SPI twice, once for the register access and another time for transmitting the command strobe can add to the variability and cause desynchronization. In order to avoid this, we exploit the multiple SPI accesses capability of the CC2420 radio which allows register access and to send strobes continuously without having to re-enable the SPI. Using this feature, we enable the SPI only once at the beginning of a channel switching operation.

We further minimize the number of in-between instructions to be executed by splitting the channel switching into two phases. In the first phase, we enable the SPI access and communicate the address of the frequency control register to the radio. In the second phase, we write into the register and transmit the command strobe to start transmit calibration. The number of in-between instructions is minimized by the fact that we overlap the first phase with the packet reception by the hardware. This way we execute only the second phase between the completion of the reception of a packet and the submission of the request for its transmission.

Accessing External Flash. Another important requirement for a dissemination protocol is that the data object has to be written into the external flash because typical sensor devices only have a small amount of RAM. In Splash, since a node is always either transmitting or receiving a packet at any given point of time, flash access has to be overlapped with a radio operation, so we write a packet to the flash while it is being transmitted by the radio. As flash access is faster than the radio transmission rate [8], the write operation completes before the radio transmission and does not cause any synchronization issues.

Handling GCC Compiler Optimizations. Although the arrival of the SFD interrupt indicating completion of the reception of a packet is synchronized across nodes, its service delay varies from node to node. The key implementation feature of Glossy is that each node executes a different number of "nop" assembly instructions based on its interrupt service delay so that all the nodes submit a request to the radio hardware at the same time for forwarding the received packet.

The most challenging problem faced during implementation is the fact that the optimization feature of the GCC compiler affects the service delay for the SFD interrupt (perhaps for some other interrupts too). Without enabling compiler optimizations, the resulting binary (a collection application coupled with Splash) was too large to fit into a sensor device. However, with optimizations enabled, minor changes to parts of the code could change the service delay, making it difficult to set the number of "nop" instructions to be executed. However, this issue can be handled as changes to the code will change the minimum duration required for servicing the SFD interrupt. While it is tedious, we can account for this change by measuring the minimum service delay after making a

change that affects the service delay. The same procedure was followed in the development of Glossy.

5 Performance Evaluation

In this section, we present the results of our evaluations carried out on the Indriya [3] and Twist [13] testbeds.

Indriya is a three-dimensional sensor network with 139 TelosB nodes spanning three floors of a building at the National University of Singapore. We compare Splash against TinyOS's DelugeT2, the de facto standard data dissemination protocol for sensor networks. For Splash, a low power setting of -10 dBm is used to build the dissemination tree and the maximum transmission power of 0 dBm is used for dissemination. For DelugeT2, we use the maximum transmission power of 0 dBm on Channel 26. We disseminate a 32-kilobyte data object for both Splash and DelugeT2.

Splash has a data payload of 64 bytes in every packet. We will show in Section 5.3 that the performance of DelugeT2 varies depending on the packet size, but there is no clear relationship between packet size and performance. Also, the impact of packet size is relatively insignificant. In this light, we adopted the default payload size of 22 bytes for DelugeT2 in our experiments on Indriya, unless otherwise stated.

The Twist sensor testbed is deployed at the Berlin University and currently it has 90 Tmote Sky devices. The experimental settings on Twist are similar to that on Indriya, except for the following differences: first, we use a lower transmission power of -15 dBm to build the dissemination tree for Splash, as Twist is a much smaller deployment than Indriya. Second, instead of using TinyOS's DelugeT2, we use Contiki's Deluge. This is because to execute TinyOS's DelugeT2, we need to execute some tools on a machine connected to base-station nodes (root nodes) which is difficult in a case of a remote testbed like Twist. We retain default settings of Contiki's Deluge including 0 dBm transmission power and Channel 26. Moreover, its default payload size of 64 bytes is also retained as Twist is a smaller deployment with stable links of good quality.

We execute Splash as a part of Contiki collection protocol [16] and Splash accesses the collection protocol's data in order to build the dissemination tree. We execute DelugeT2 as a part of TinyOS collection protocol, CTP [11] by coupling the DelugeT2 with the TinyOS's standard "TestNetwork" application with its default settings. We also compare Splash against DelugeT2 running as a standalone golden image (GI) without CTP. Note that the standalone version is seldom used in practice, as a dissemination protocol is only useful when coupled with a real application.

5.1 Summary of Testbed Results

The summary of our results on Indriya and Twist are shown in Tables 2 and 3 respectively. For each experimental run, we randomly picked a node as the root of the dissemination tree. In the tables, "size" indicates the depth of the Splash's dissemination tree, and R1, R2 and R3 indicate the average *reliability* per node after the first, second and third rounds of dissemination respectively. By reliability, we refer to the fraction of the data object that has been successfully downloaded by a node. $N_{R3-100\%}$ is the proportion of nodes that have 100% of the disseminated data object after the third round. Recall that XOR coding is employed in the third dissemination round. R_{lr} indicates the average reliability per node after local recovery. T_{Splash} is the time taken for Splash to complete the dissemination, i.e. when *every* node in the network has successfully downloaded the entire data object. Similarly, $T_{DelugeT2+CTP}$, $T_{DelugeT2GI}$, and T_{Deluge} are the corresponding times taken for DelugeT2 with CTP, DelugeT2 as standalone golden image, and Contiki's Deluge respectively, to complete the dissemination.

Indriya Testbed. We observe from Table 2 that on average Splash takes about 25 seconds (see T_{Splash}) to complete the dissemination of a 32-kilobyte object, while DelugeT2 coupled with CTP takes about 524 seconds. Splash reduces dissemination time by an average factor of 21.06 (93.68% reduction). Splash also outperforms DelugeT2 running as a standalone golden image by a factor of 12.43 (89.2% reduction). One obvious drawback of DelugeT2 is that there is a large variation in its dissemination time, ranging from 209 seconds to 1300 seconds. This is likely due to variations in the conditions of the default Channel 26 since DelugeT2 uses a fixed channel. By using multiple rounds of dissemination, opportunistic overhearing, and channel cycling, Splash is more resilient to variations in the channel conditions. In particular, a node in Splash has the potential to receive a packet up to 6 times, and more importantly, on up to 6 different channels. If the quality of one or two channels is bad, a packet can potentially be successfully decoded on one of the other remaining channels.

We also observe that the dissemination time for DelugeT2 as golden image is usually less than DelugeT2 with CTP. This is because dissemination traffic in the latter case has to contend with CTP's application traffic. While Splash relies on Contiki's Collection Protocol to build its dissemination tree, like Glossy [9], Splash disables all the interrupts other than the Start Frame Delimiter interrupt during its three rounds of dissemination where constructive interference is exploited. This means that any underlying application will be temporarily suspended and most of the Splash's traffic will be served exclusively without interference from any application traf-

Table 2: Summary of results for 139-node Indriya testbed.

		Splash						DelugeT2	
Tree No.	size [hops]	R1 [%]	R2 [%]	R3 [%]	$N_{R3-100\%}$ [%]	R_{lr} [%]	T_{Splash} [sec]	$T_{DelugeT2+CTP}$ [sec]	$T_{DelugeT2GI}$ [sec]
1	5	84.54	97.23	98.47	91.30	100.00	22.49	1300	924
2	6	86.52	96.91	98.58	92.03	100.00	22.61	286	160
3	7	76.68	94.62	97.80	86.23	100.00	23.18	209	286
4	7	88.02	96.12	97.78	92.75	100.00	23.74	218	158
5	9	76.97	93.65	96.69	81.88	100.00	23.86	649	180
6	7	76.73	95.27	98.16	89.86	100.00	25.98	610	160
7	7	80.75	93.51	96.98	89.13	100.00	26.25	365	379
8	7	83.57	94.43	96.01	87.68	100.00	26.89	377	277
9	5	82.46	95.26	97.47	85.51	100.00	28.09	676	313
10	8	84.28	94.92	96.70	86.23	100.00	28.39	550	216
Average		82.05	95.19	97.46	88.26	100.00	25.15	524	305.3

Table 3: Summary of results for 90-node Twist testbed.

		Splash						Deluge
Tree No.	size [hops]	R1 [%]	R2 [%]	R3 [%]	$N_{R3-100\%}$ [%]	R_{lr} [%]	T_{Splash} (for a 32KB file) [sec]	T_{Deluge} (for a 2KB file) [sec]
1	4	90.58	97.09	99.22	94.38	100.00	20.07	356.60
2	4	81.08	94.70	99.31	92.13	100.00	20.19	431.48
3	4	86.53	96.19	98.00	91.01	100.00	22.79	351.67
4	4	78.64	94.10	98.12	84.09	100.00	23.37	518.19
5	4	81.42	93.95	97.98	89.89	100.00	23.41	467.00
6	4	78.04	93.55	96.82	85.39	100.00	26.66	439.81
7	4	83.90	95.18	97.54	89.89	100.00	26.79	345.28
8	4	83.70	93.64	96.45	84.27	100.00	27.32	388.68
9	6	81.58	93.35	97.02	85.39	100.00	27.45	484.10
10	5	80.78	93.09	97.11	85.39	100.00	29.25	397.59
Average		82.62	94.48	97.76	88.18	100.00	24.73	418.04

fic. On the other hand, because DelugeT2 is built on TinyOS services, it is not possible to completely disable all the interrupts during its execution. DelugeT2 as golden image provides us with the baseline performance without interference from application traffic. Note that application suspension in Splash is not a problem as most sensor applications have no real-time requirements. Moreover, interrupts are re-enabled long before the completion of dissemination, before starting the round of local recovery that dominates the dissemination time (see Fig. 7). Applications are suspended for only about 8.2 seconds while disseminating the 32-kilobyte object.

Twist Testbed. As shown in Table 3, Splash's performance on Twist is similar to that on Indriya. It takes about 25 seconds on average to complete the dissemination of a 32-kilobyte object. On the other hand, because the Contiki implementation of Deluge is less efficient, it takes about 418 seconds to disseminate a much smaller object of 2 kilobytes. Note that Contiki Deluge is a thin implementation with minimal functionality that allows only minimal changes to its settings. Hence, Splash is able to significantly outperform Contiki's Deluge even when disseminating a data object that is 16 times larger. Splash effectively achieves a *network-wide* goodput of above 10 kilobits/sec per node on both Indriya and Twist testbeds, which is higher than that of all existing network-wide data dissemination protocols [6, 5, 7, 10, 12, 30, 18, 14, 17] in the literature.

Memory Consumption. Splash not only outperforms DelugeT2 in terms of speed, it is also much more efficient than DelugeT2 in terms of memory usage. Splash requires only 11.38 kilobytes of ROM and 0.13 kilobytes of RAM whereas DelugeT2 requires 21.01 and 0.81 kilobytes of ROM and RAM respectively. Hence, Splash uses 9.63 kilobytes of ROM and 0.68 kilobytes of RAM less than DelugeT2. Given that it is not uncommon for sensor devices to have only about 48 and 10 kilobytes of ROM and RAM respectively, these are significant savings in memory, that will be available for use by sensor applications.

Comparison to Existing Protocols. Because we were not able to obtain the code for the state-of-the-art dissemination protocols ECD [6] and MT-Deluge [10], we used an indirect method to compare Splash against them and other existing dissemination protocols [5, 12, 30, 18]. It turns out that these protocols are all evaluated against Deluge and so we have a convenient common baseline with which to compare against without having to implement and evaluate them individually. We present the relative performance of Splash to these protocols in Ta-

Table 4: Comparison of Splash to existing protocols.

Protocol	No. of nodes	File size [KB]	Reduction factor
MNP ([18], 2005)	100	5	1.21
MC-Deluge ([30], 2005)	25	24.3	1.6
Rateless Deluge ([12], 2008)	20	0.7	1.47
ReXOR ([5], 2011)	16	4	1.53
ECD ([6], 2011)	25	10	1.44
MT-Deluge ([10], 2011)	20	0.7	2.42
Splash	139	32	21

ble 4. In the fourth column, we present the reduction factor achieved by each of these algorithms compared to Deluge. It is evident that Splash's performance is significantly better than that of the state-of-the-art protocols. Not only is Splash faster by an order of magnitude, but we also achieve this improvement on a larger testbed and with a bigger file than all the previous algorithms. Note also that most of the results for the existing protocols in Table 4 are compared against classical Deluge (Deluge 2.0 of TinyOS 1), which is in fact slower than DelugeT2, against which we have compared Splash.

Energy Consumption. Duty cycling is typically adopted by applications that transmit a data packet once in a while, and not for dissemination that involves transfer of large amounts of data [25]. As duty-cycled transmissions involve a large overhead such as the transmission of a long preamble before sending every packet [22], they make dissemination significantly more expensive in terms of both time and energy. This drives most of the dissemination protocols in the literature [14, 12, 5, 6, 10, 30] to keep the radio awake during dissemination as required in Splash. Therefore, energy consumption is directly proportional to the dissemination time. This means Splash reduces energy consumption by the same factor by which it reduces dissemination time.

5.2 Contribution of Individual Techniques

In order to achieve a high reliability, Splash incorporates four key techniques: (1) XOR coding; (2) transmission density diversity; (3) opportunistic overhearing; and (4) channel cycling. We now evaluate the contribution of these techniques together with local recovery.

XOR Coding. We employ XOR coding in the third round of dissemination. The goal of using XOR coding is to significantly increase the number of nodes that successfully receive the entire file so that local recovery will be much more efficient. We present the proportion of nodes that achieve a reliability of 100% before and after the third round of XORed dissemination on Indriya in Table 5. The largest improvement was observed for the fifth tree where the use of XOR coding increases the percentage of nodes having the full object from 9.42% to 81.88%. On average, the number of nodes with the full data object is more than doubled. Similar results were observed on the Twist testbed.

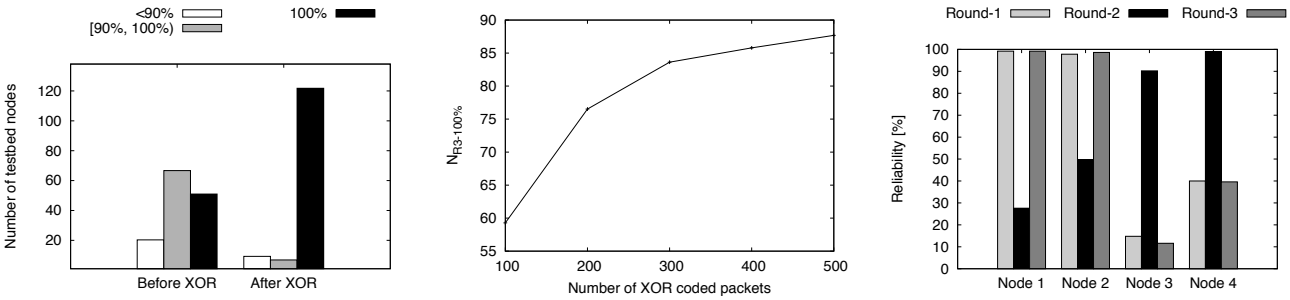
Table 5: Proportion of nodes with 100% reliability before and after the third round of XOR coding on Indriya.

Tree No.	Before XOR	After XOR
1	57.25	91.30
2	50.72	92.03
3	21.74	86.23
4	33.33	92.75
5	9.42	81.88
6	26.09	89.85
7	23.91	89.13
8	47.10	87.68
9	51.45	85.51
10	48.55	86.23
Avg.	36.96	88.26

To validate our hypothesis that XOR's effectiveness comes from helping the nodes that already have most of the packets, we plot in Fig. 5(a) the average number of nodes per tree found in the three different bins of reliability for Indriya, namely <90%, between 90% and 100%, and 100%. We see that before the third dissemination round, there are about 20 nodes in the first bin with reliability less than 90% and 67 nodes in the second bin with reliability between 90% and 100%. XOR coding is able to move most of these nodes in the first 2 bins into the third bin with 100% reliability. In particular, XOR coding can reduce the size of the second bin from 67 to 7, to give a total of 122 nodes in the 100% bin. Similar results were observed on the Twist testbed.

For the 32-kilobyte file that we used in our experiments, we XOR coded and transmitted each of the 500 packets (with a packet payload size of 64 bytes) constituting the file. One pertinent question is whether we can do with fewer packets since an XORed packet already contains the information of 20 packets. In Fig. 5(b), we present a plot of $N_{R3-100\%}$ against the number of XOR coded packets transmitted, averaged over five experimental runs on different dissemination trees. Note that only about 37% of the nodes have downloaded the whole file after the first two rounds of dissemination. It is clear from Fig. 5(b) that 100 packets is not enough, and that there is a significant improvement in $N_{R3-100\%}$ as we transmit more coded packets until about 400 packets. From 400 to 500 packets, we obtain only a small increase of about 2% in $N_{R3-100\%}$ (about 3 nodes). While the improvement is small, since local recovery over CSMA/CA can be expensive, we decide to transmit all the 500 coded packets for completeness since the extra 100 transmissions take only an extra 0.56 seconds.

Transmission Density Diversity. To understand the effectiveness of our attempt to exploit transmission density diversity, we disseminate a 32-kilobyte data object without the leaf nodes transmitting (*Round-1*). Imme-



(a) Distribution of avg. no. nodes across three reliability bins. (b) $N_{R3-100\%}$ Vs. no. of XOR coded transmissions. (c) Effectiveness of transmission density diversity.

Figure 5: Contributions of XOR coding and transmission density diversity.

Table 6: Performance of Splash with and without opportunistic overhearing.

	With overhearing			Without overhearing		
No.	N_{lrpkts}	$N_{R3-100\%}$	T_{Splash} [sec]	N_{lrpkts}	$N_{R3-100\%}$	T_{Splash} [sec]
1	1860	78.99	28.28	5536	79.71	44.07
2	1433	89.13	23.64	2415	84.06	36.19
3	1876	89.13	27.00	2531	85.51	34.98
4	420	93.48	21.94	1529	90.58	24.73
5	1356	90.58	22.68	1131	83.33	26.75
Avg.	1389	88.26	24.71	2628.4	84.64	33.34

diately after that, the object is disseminated again but with all the nodes transmitting (*Round-2*). Finally, we repeated the transmission without the leaf nodes transmitting (*Round-3*). This approach allows us to determine whether a node gains from a low transmission density or a node gains from a high transmission density. The same channel assignment is used for all three rounds.

We run this experiment five times on a dissemination tree. As an illustration, we present the reliability observed on four nodes in each of the three rounds of an experimental run in Fig. 5(c). Nodes 1 and 2 benefit from a low transmission density (without leaves) as the achieved reliability is higher in the first and third rounds of dissemination. On the other hand, nodes 3 and 4 benefit from a high transmission density with all nodes transmitting. On average, we found that 38.7% of the nodes benefit from a low transmission density and achieve higher reliability than that for the higher transmission density. The proportion of nodes that benefit from a high transmission density is lower, about 18.1% achieve higher reliability at the higher transmission density compared to that for the lower transmission density. Nevertheless, the key insight is that by varying the number of transmitters between transmission rounds, different sets of nodes will correctly decode packets over different transmission rounds.

Opportunistic Overhearing. Table 6 compares the performance of Splash with and without opportunistic overhearing on five dissemination trees on Indriya. The table shows the total number of packets to be recovered during local recovery (N_{lrpkts}) together with $N_{R3-100\%}$ and T_{Splash}. We found that T_{Splash} is increased by 8.6 seconds on average when opportunistic overhearing is not employed. Quite clearly, this is because the number of corrupted/missed packets N_{lrpkts} is typically larger when there is no overhearing, as observed on the first four of the five considered trees. In the case of the fifth tree, we found that overhearing did not lead to a smaller number of corrupted/missed packets N_{lrpkts}. However, Splash with overhearing is still faster because the proportion of nodes that have downloaded the full data object after 3 dissemination rounds ($N_{R3-100\%}$) is larger. In other words, overhearing helps not just by increasing the likelihood that packets are transmitted successfully, it also helps by ensuring that more nodes have downloaded the complete file.

Channel Cycling. In order to evaluate the effectiveness of channel cycling, we compare Splash with channel cycling against Splash without channel cycling i.e., by using the same channel assignment in all three dissemination rounds. We plot the resulting performance for five dissemination trees on Indriya in Table 7. Without channel cycling, there is a drop in both reliability (R3) and the percentage of nodes having the full data object after the third round of dissemination ($N_{R3-100\%}$). In addition to better average-case performance, we also see that channel cycling can significantly reduce the variance in performance. We see that T_{Splash} varies between 22.49 s and 28.39 s with channel cycling, while it varies between 26.24 s and 45.08 s without.

Local Recovery. After three rounds of dissemination, about 88% of the nodes would have successfully received the entire data object on average on both of the testbeds (see Column $N_{R3-100\%}$ in Tables 2 and 3). In Fig. 6, we plot the CDF of the reliability of those nodes that did not successfully receive the complete file after three rounds of dissemination. We see that among these nodes, only about 3% and 1% have less than 10% of the data on Indriya and Twist respectively. About 40% have at least 90% of the data object. In Fig. 7, we present the

Table 7: Performance of Splash with and without channel cycling.

	With cycling			Without cycling		
No.	R3	$N_{R3-100\%}$	T_{Splash} [sec]	R3	$N_{R3-100\%}$	T_{Splash} [sec]
1	96.98	89.13	26.25	92.33	76.81	45.08
2	98.16	89.86	25.98	95.56	86.23	26.24
3	96.69	81.88	23.86	92.15	73.19	34.79
4	98.47	91.30	22.49	91.86	79.71	34.58
5	96.70	86.23	28.39	95.61	85.51	31.51
Avg.	97.40	87.68	25.39	93.50	80.29	34.44

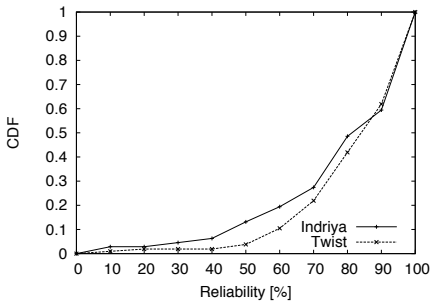


Figure 6: Distribution of the reliability of nodes with reliability less than 100%.

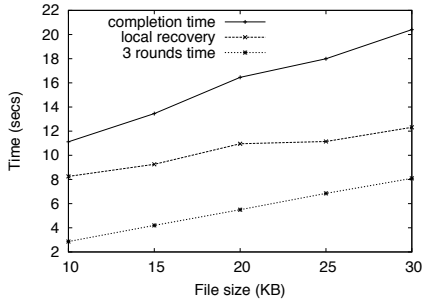


Figure 7: Breakdown of completion time for different file sizes.

time taken for local recovery for data objects of different sizes on Indriya. We also present the time taken for the first three rounds of dissemination and the completion time on the same graph. As expected, the time spent in the first three rounds increases linearly with the object size whereas time taken for local recovery is not strictly linear due to the variations in the number of packets to be recovered and the randomness involved in CSMA/CA.

5.3 Effect of Packet Size

It is well-known that the reliability of constructive interference decreases as packet size increases [9, 28]. To justify our choice of 64 bytes for the Splash payload, we compare the performance of Splash for the default payload size against the maximum possible payload size of 117 bytes (which results in a maximum-sized packet

Table 8: Performance of Splash for two different payload sizes.

64 bytes				117 bytes			
R1	R2	R3	$N_{R3-100\%}$	R1	R2	R3	$N_{R3-100\%}$
85.12	96.82	98.68	92.03	78.19	91.60	94.47	78.26
86.35	96.64	98.30	91.30	80.58	92.04	93.52	78.99
89.41	96.90	98.83	93.48	81.91	94.65	96.45	82.61
84.64	96.20	97.67	88.41	78.96	92.59	95.20	82.61
84.49	96.99	98.29	89.13	72.08	87.54	90.35	70.29
86.00	96.71	98.35	90.87	78.34	91.68	94.00	78.55

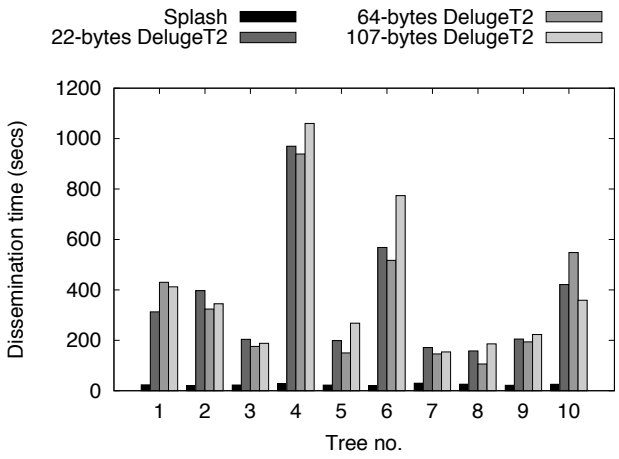


Figure 8: Comparison of Splash against DelugeT2 configured with different payload sizes on Indriya.

of 128 bytes) for five dissemination trees on Indriya in Table 8. As expected, reliability decreases with the larger payload size, so we set the default payload size for Splash to 64 bytes.

It is known that the performance of DelugeT2 varies with packet size [2], so in order to compare Splash fairly to DelugeT2, we also investigated the performance of DelugeT2 for different payload sizes. We constructed 10 random dissemination trees on Indriya, and on each of them we disseminated a 32-kilobyte object using Splash and DelugeT2 configured with payload sizes of 22 bytes (default), 64 bytes, and the maximum value of 107 bytes. We ensured that Splash and the three versions of DelugeT2 were executed back-to-back on each of the dissemination trees so as to minimize the temporal variations in channel conditions across these executions. The results are shown in Fig. 8. For DelugeT2, we found that while there was some variation in the average dissemination times depending on the payload size and the payload size that achieves the best performance depends on the actual network conditions, the differences in performance are not significant, at least not when compared to the dissemination times achieved by Splash.

6 Conclusion

We propose Splash, a fast and scalable dissemination protocol for wireless sensor networks, that exploits constructive interference and channel diversity to achieve speed and scalability. To achieve high reliability, Splash incorporates the use of transmission density diversity, opportunistic overhearing, channel-cycling, and XOR coding. We demonstrated with experiments on two large multihop sensor networks that Splash can achieve an order of magnitude reduction in dissemination time compared to state-of-the-art dissemination protocols.

Acknowledgements

We would like to thank the anonymous reviewers and our shepherd, Rodrigo Fonseca for their valuable comments and suggestions. This work was partially supported by the NRF Singapore through the SMART (R-252-002-430-592) program.

References

[1] ALIZAI, M. H., LANDSIEDEL, O., LINK, J. A. B., GOTZ, S., AND WEHRLE, K. Bursty Traffic over Bursty Links. In *Proceedings of SenSys* (2009).

[2] C., R. K., SUBRAMANIAN, V., ULUAGAC, A. S., AND BEYAH, R. SIMAGE: Secure and Link-Quality Cognizant Image Distribution for Wireless Sensor Networks. In *Proceedings of GLOBECOM* (2012).

[3] DODDAVENKATAPPA, M., CHAN, M. C., AND ANANDA, A. Indriya: A Low-Cost, 3D Wireless Sensor Network Testbed. In *Proceedings of TRIDENTCOM* (2011).

[4] DODDAVENKATAPPA, M., CHAN, M. C., AND LEONG, B. Improving Link Quality by Exploiting Channel Diversity in Wireless Sensor Networks. In *Proceedings of RTSS* (2011).

[5] DONG, W., CHEN, C., LIU, X., BU, J., AND GA, Y. A Lightweight and Density-Aware Reprogramming Protocol for Wireless Sensor Networks. In *IEEE TRANSACTIONS ON MOBILE COMPUTING* (2011).

[6] DONG, W., LIU, Y., WANG, C., LIU, X., CHEN, C., AND BU, J. Link Quality Aware Code Dissemination in Wireless Sensor Networks. In *Proceedings of ICNP* (2011).

[7] DONG, W., LIU, Y., WU, X., GU, L., AND CHEN, C. Elon: Enabling Efficient and Long-Term Reprogramming for Wireless Sensor Networks. In *Proceedings of SIGMETRICS* (2010).

[8] DUQUENNOY, S., ÖSTERLIND, F., AND DUNKELS, A. Lossy Links, Low Power, High Throughput. In *Proceedings of SenSys* (2011).

[9] FERRARI, F., ZIMMERLING, M., THIELE, L., AND SAUKH, O. Efficient Network Flooding and Time Synchronization with Glossy. In *Proceedings of the IPSN* (2011).

[10] GAO, Y., BU, J., DONG, W., CHEN, C., RAO, L., , AND LIU, X. Exploiting Concurrency for Efficient Dissemination in Wireless Sensor Networks. In *Proceedings of DCOSS* (2011).

[11] GNAWALI, O., FONSECA, R., JAMIESON, K., MOSS, D., AND LEVIS, P. Collection Tree Protocol. In *Proceedings of SenSys* (2009).

[12] HAGEDRON, A., STAROBINSKI, D., AND TRACHTENBERG, A. Rateless Deluge: Over-the-Air Programming of Wireless Sensor Networks using Random Linear Codes. In *Proceedings of IPSN* (2008).

[13] HANDZISKI, V., KOPKE, A., WILLIG, A., AND WOLISZ, A. TWIST: A Scalable and Reconfigurable Testbed for Wireless Indoor Experiments with Sensor Network. In *Proceedings of REALMAN* (2006).

[14] HUI, J. W., AND CULLER, D. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *Proceedings of SenSys* (2004).

[15] KIM, S., PAKZAD, S., CULLER, D. E., DEMMEL, J., FENVES, G., GLASER, S., AND TURON, M. Health Monitoring of Civil Infrastructures Using Wireless Sensor Networks. In *Proceedings of IPSN* (2007).

[16] KO, J., ERIKSSON, J., TSIFTES, N., DAWSON-HAGGERTY, S., DURVY, M., VASSEUR, J., TERZIS, A., DUNKELS, A., AND CULLER, D. Beyond Interoperability: Pushing the Performance of Sensor Network IP Stacks. In *Proceedings of SenSys* (2011).

[17] KULKARNI, S. S., AND ARUMUGAM, M. INFUSE: A TDMA based Data Dissemination Protocol for Sensor Networks. Tech. rep., Michigan State University, 2004.

[18] KULKARNI, S. S., AND WANG, L. MNP: Multihop Network Reprogramming Service for Sensor Networks. In *Proceedings of ICDCS* (2005).

[19] LEVIS, P., AND CULLER, D. Mate: a Virtual Machine for Tiny Networked Sensors. In *Proceedings of ASPLOS* (2002).

[20] LEVIS, P., PATEL, N., CULLER, D., AND SHENKER, S. Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *Proceedings of NSDI* (2004).

[21] LIANG, C.-J. M., PRIYANTHA, N. B., , LIU, J., AND TERZIS, A. Surviving Wi-Fi Interference in Low Power ZigBee Networks. In *Proceedings of SenSys* (2010).

[22] MOSS, D., AND LEVIS, P. BoX-MACs: Exploiting Physical and Link Layer Boundaries in Low-Power Networking. Tech. rep., Technical Report SING-08-00, Stanford University, 2008.

[23] OSTERLIND, F., AND DUNKELS, A. Approaching the Maximum 802.15.4 Multihop Throughput. In *Proceedings of HotEmNets* (2008).

[24] RAMAN, B., CHEBROLU, K., BIJWE, S., AND GABALE, V. PIP: A Connection-Oriented, Multi-Hop, Multi-Channel TDMA-based MAC for High Throughput Bulk Transfer. In *Proceedings of SenSys* (2010).

[25] ROSSI, M., BUI, N., ZANCA, G., STABELLINI, L., CREPALDI, R., AND ZORZI, M. SYNAPSE++: Code Dissemination in Wireless Sensor Networks Using Fountain Codes. In *IEEE TRANSACTIONS ON MOBILE COMPUTING* (2010).

[26] SRINIVASAN, K., JAIN, M., CHOI, J. I., AZIM, T., KIM, E. S., LEVIS, P., AND KRISHNAMACHARI, B. The K-Factor: Inferring Protocol Performance Using Inter-link Reception Correlation. In *Proceedings of Mobicom* (2010).

[27] WANG, Q., ZHU, Y., AND CHENG, L. Reprogramming Wireless Sensor Networks: Challenges and Approaches. In *IEEE Network Magazine* (2006).

[28] WANG, Y., HE, Y., MAO, X., LIU, Y., HUANG, Z., AND YANG LI, X. Exploiting Constructive Interference for Scalable Flooding in Wireless Networks. In *Proceedings of INFOCOM* (2012).

[29] WU, Y., STANKOVIC, J. A., HE, T., LU, J., AND LIN, S. Realistic and Efficient Multi-Channel Communications in Wireless Sensor Networks. In *Proceedings of INFOCOM* (2008).

[30] XIAO, W., AND STAROBINSKI, D. Poster Abstract: Exploiting Multi-Channel Diversity to Speed Up Over-the-Air Programming of Wireless Sensor Networks. In *Proceedings of SenSys* (2005).

Expanding Rural Cellular Networks with Virtual Coverage

Kurtis Heimerl
kheimerl@cs.berkeley.edu
UC Berkeley

Kashif Ali
kashif@cs.berkeley.edu
UC Berkeley

Joshua Blumenstock
joshblum@uw.edu
University of Washington

Brian Gawalt
bgawalt@eecs.berkeley.edu
UC Berkeley

Eric Brewer
brewer@cs.berkeley.edu
UC Berkeley

Abstract

The cellular system is the world's largest network, providing service to over five billion people. Operators of these networks face fundamental trade-offs in coverage, capacity and operating power. These trade-offs, when coupled with the reality of infrastructure in poorer areas, mean that upwards of a billion people lack access to this fundamental service. Limited power infrastructure, in particular, hampers the economic viability of wide-area rural coverage.

In this work, we present an alternative system for implementing large-scale rural cellular networks. Rather than providing constant coverage, we instead provide *virtual coverage*: coverage that is only present when requested. Virtual coverage powers the network on-demand, which reduces overall power draw, lowers the cost of rural connectivity, and enables new markets.

We built a prototype cellular system utilizing virtual coverage by modifying a GSM base station and a set of Motorola phones to support making and receiving calls under virtual coverage. To support the billions of already-deployed devices, we also implemented a small radio capable of adding backwards-compatible support for virtual coverage to existing GSM handsets. We demonstrate a maximum of 84% power and cost savings from using virtual coverage. We also evaluated virtual coverage by simulating the potential power savings on real-world cellular networks in two representative developing counties: one in sub-Saharan Africa and one in South Asia. Simulating power use based on real-world call records obtained from local mobile operators, we find our system saves 21-34% of power draw at night, and 7-21% during the day. We expect even more savings in areas currently off the grid. These results demonstrate the feasibility of implementing such a system, particularly in areas with solar or otherwise-intermittent power sources.

1 Introduction

No recent technology has had a greater impact on economic development than mobile phones, which comprise the largest networks on Earth and cover over five billion subscribers [18]. Unfortunately, many people still lack this fundamental service. Although it is difficult to know the total number of potential users currently without coverage, it is likely more than a billion.

Nearly 95% of this uncovered population live in rural areas without grid power [12]. The primary reason for their lack of service is economic; operators are unwilling to make the large infrastructure investments (or pay the large operating costs) required to operate in areas without enough users to cover expenses. Contacts in rural areas have reported prices between five hundred thousand to one million USD for the installation of a cell tower in an area without existing power or network [25]. Our theory of change is simple: by reducing the cost of infrastructure we make mobile phones viable for new areas and users. With lower costs, we believe operators will naturally extend their reach into more rural areas, and new rural entrants become economically viable.

Recent advances have dramatically lowered the price of cellular equipment [23] and backhaul networks [26]. Unfortunately, power remains a fundamental cost in any deployment, dominating both the capital and operating costs of rural cellular networks. The International Telecommunications Union has indicated that 50% of the OPEX cost for a rural network is power [17]. Commercial equipment providers have attempted to address this fact [2, 28, 33], reducing the power consumption of small-scale cellular equipment to less than 90 watts. Unfortunately, further reducing power draw is impossible without causing service interruptions; the power amplifier quickly dominates total power used (65%-84% of total draw) and is directly tied to coverage radius and capacity. Each watt of power drawn is another watt needing generation (usually diesel [12]) and

storage; a severe limitation with expensive rural power.

One obvious way to save more power is to turn off portions of the network for a period of time (typically at night). This is common in rural and developing regions [17]. For instance, in the Punjab province of Pakistan, citizens were without grid power for over eighteen hours a day [1]. Unfortunately, this means that users cannot make important calls (such as emergency calls, a critical use case [15]) while the network is off.

Our solution, *virtual coverage*, resolves this concern. Virtual coverage also powers down individual cellular towers, but only when not in use, which we demonstrate to be a substantial fraction of the time. When network is needed, as signaled by a user initiating or receiving a call, the power is restored and the tower is available for communications. This design allows us to save a large amount of power in the largest networks on Earth while still providing consistent coverage at all times.

Specifically, individual cellular towers are powered-down (i.e., in "idle" mode) during periods of prolonged idleness. Although powering down a tower is conceptually simple, waking one is much harder; when a user wishes to make an outbound call, they are able to wake the network by sending a burst from either a modified GSM phone (our Wake-up Phone) or via a small, low-cost, push-button transmitter (our Wake-up Radio, functionally similar to a garage-door opener). Mobile-terminated calls require no changes to user behavior; the tower simply turns on and holds the call until the requested user connects. This solution also benefits from economies of scale by leveraging existing handset and radio equipment.

We implemented this design by modifying a Range Networks 5150 2G GSM base transceiver station (BTS). We first show that "idle" mode saves between 65% to 84% of the power on a BTS, depending on the coverage and capacity required. We also demonstrate that the user's experience is not dramatically affected, with the setup of all calls increasing by only two seconds for the Wake-up Phone and an average of at most 25 seconds in standard, unmodified phones using the Wake-up Radio. We show that an installation using virtual coverage in a low-density area could operate with less than one-sixth of the solar panels, batteries, and price of a traditional setup. Lastly, we demonstrate that a virtual coverage network's power requirements scale sub-linearly with the total number of calls. This allows smaller operators to invest in their network as it grows, rather than with one large capital expenditure, reducing their risk.

In addition, we simulated the use of our technology in two existing developing world cellular networks. We gathered a week of tower-level call activity from one country in sub-Saharan Africa (roughly 15 million calls) and one country in South Asia (roughly 35 million calls). Using these records, we calculated the exact amount of "idle time" (time where no calls were active) in each network and combined them with estimates of the power draw for each tower. We show that, by utilizing virtual coverage, we could reduce the network power draw by 34% at night (21% during the day) for our South Asian operator. In sub-Saharan Africa, where towers are more heavily utilized, we are able to save 21% of the power at night (and just 7% during the day). Although this simulation uses only existing networks, we expect calling patterns in currently unserved areas to have more available idle time and thus, power savings. This reduction in operating power would dramatically reduce the operating cost of an off-grid, renewable powered, rural wide-area cellular network, enabling cheaper, greener telecommunications for people currently without network connectivity.

Summary of contributions:

- The concept of *Virtual Coverage*: on demand wide-area cellular networking;
- A working cellular base station implementing virtual coverage using OpenBTS;
- The implementation of a handset capable of waking a virtual coverage enabled BTS;
- The design and implementation of a low-cost wake-up radio capable of waking the network, for supporting existing handsets;
- A technical evaluation of virtual coverage, showing "idle" power savings, user experience impact, and sub-linear growth in power needs as a function of total calls; and
- An *in situ* evaluation of the potential power savings based on trace data from all the cellular towers operated by both an sub-Saharan African and South Asian telecommunication firm.

2 Design

Our system is targeted at a specific set of users: those in rural areas too sparse or poor to support traditional cellular infrastructure. This encompasses more than a billion people, in both developed and developing areas. We start with use cases that inform our design.

Alaska Large swaths of rural America live in areas without network coverage. Often these are agricultural or mountainous areas where the population density is too low or coverage too difficult to warrant the (expensive) deployment. One example is Hatcher Pass in South Central Alaska. Though surrounded by large towns (Palmer, Wasilla, and Houston), the mountains are too far from these population centers for network coverage,

especially in the valleys that define the range.

Our system would enable lightweight, autonomous, solar-powered cellular equipment that can be easily deployed on a central mountain top, providing "on demand" network coverage for travelers in need of communication. As communications in these areas will be rare, we want to enable wide-area coverage, amortizing the cost of deploying and maintaining the cellular equipment.

Uganda Another primary use case is in the developing world, where potentially billions of wireless consumers live in areas without coverage. This is almost always in rural areas, as the high population density of cities (despite the low per capita earnings) ensures that wireless providers will see a return on their investment. Similarly, even though the population in rural areas may be higher in the developing world than in the rural U.S., their lower income discourages investment.

One potential target for virtual coverage is rural Uganda. One example is city of Mpigi, an hour from the capital of Kampala. As a major trading hub, multiple carriers provide coverage in the center of town. However, in the hills less than 20 kilometers away, there's no coverage at all. Heimerl et al. [15] showed how populations in these areas manage to communicate with their limited cellular coverage. They found that users heavily valued emergency services, and were unwilling to accept a fully asynchronous telecommunication system for this reason. Thus we aim for low-cost but continuous operation, explicitly supporting emergency services.

2.1 Design Goals

With the above use cases in mind, we have generated a set of design goals:

- Enable solar-powered cellular infrastructure capable of wide-area coverage;
- Provide continuous network availability (with a small start-up delay) to support emergency communications;
- Reduce infrastructure cost, enabling coverage for areas currently too poor or sparse for traditional cellular; and
- Utilize existing economies of scale by building off of existing GSM handsets and base stations and minimizing hardware changes.

We also note that our design explicitly does not support mobile handoff, nor does it ensure that wake-up requests are made in good faith. We address these concerns in Section 7.

3 Background

Cellular telephony is an enormous field, with multiple standards deployed across nearly every nation. In this section, we detail the specific wireless standards and hardware most suited to virtual coverage.

3.1 Cellular Telephony

The 2G (GSM) standard was officially launched in Finland in 1991. Subsequent 3G (UMTS) and 4G (LTE) standards first appeared in 2001 and 2009, respectively. As the standards progressed, the effective bit-rate for channels increased, primarily to support high-bandwidth data services such as streaming video.

Unfortunately, these superior encodings are more sensitive to errors and loss [29], which limits their propagation and usefulness in rural areas. With this in mind, most 3G/4G deployments are smaller in scale, primarily targeting dense urban areas. The 2G GSM standard, especially in the lower 900-MHz band, delivers the most consistent network propagation.

For these reasons, the 2G standard is still present in almost all commercial cellular hardware. Similarly, it is also present in almost all cellular handsets, with many brand new handsets in the developing world supporting only the 2G standard.

There have been recent advances in open-source GSM telephony, specifically the OpenBTS [23] project. This is an open implementation of the 2G GSM standard. As such, this is the wireless technology we use to prototype virtual coverage. However, there is no fundamental difficulty applying Virtual Coverage to 3G/4G networks.

3.2 Cellular Base-Station Hardware

In a modern software-defined GSM base station there are three core pieces of hardware that draw power, while the rest is passive. These pieces are the computer, the radio, and the power amplifier. Figure 1 shows our base station.

Unlike many other wireless systems (like 802.11), GSM coverage range is inherently limited by the *uplink* power (phone to BTS) and not the tower's transmit power. The handset must be able to reach the tower, and increasing our broadcast power does not make that any easier. Though there is equipment on the BTS that improves this slightly, any transmit amplifications over 10W will not improve the range of the system; the GSM standard sets the maximum handset power to be 2W.

However, past the 10W limit, increasing the transmit power does allow for *more* communications at the same range. The extra power can amplify other channels, increasing the totally capacity of the tower. Table 1 provides an example.

	Range (km)	Capacity (Calls)
2W Tower	7	7
10W Tower	35	7
50W Tower	35	35

Table 1: Range Networks [28] cellular tower propagation and capacity specifications.

In any such wide-area setup (greater than 1 kilometer), the amplifier will dominate the power consumed by the unit. A 10W amplifier draws 45W of power (65% of the total) in a low-capacity BTS. A 50W amplifier draws 130W (84% of the total) for a high-capacity, 35 concurrent call, BTS. This amplifier operates continuously, amplifying the beacon channel.

Because of these properties, any attempt to save power in a wide-area cellular network must change the behavior of the power amplifier [3]. Unfortunately, this will also affect the user experience; amplification is the mechanism by which the tower broadcasts long distances and increases capacity. This is the core problem we address with virtual coverage: meaningfully covering a sparse population is currently energy-intensive.

4 System Implementation

Enabling virtual coverage requires a holistic rethinking of the base station itself. First, the BTS must be modified to enable programmatic control of the power amplifier. This will allow us to enter an "idle" mode in which the power amplifier is turned off. As a byproduct of this, the network is unavailable during this period.

Second, we must implement a mechanism for allowing users to wake the cellular tower remotely and promptly, thus enabling coverage on demand. We implemented two models of virtual coverage wake up: 1) implementing software-only handset modifications to send special wake-up bursts, and 2) developing a custom autonomous low-cost radio that sends the same message, allowing the system to work with existing, unmodified handsets. After detection of this burst, the network exits the idle state and resumes normal operation.

4.1 Enabling Low-Power Modes in Cellular Infrastructure

Virtual coverage requires the base station to have a low-power mode when the network is not in use. There are two core changes needed create a low-power mode for a GSM base station. First, the hardware must be modified to provide a mechanism for programmatic control of the power amplifier, the primary power draw. Second, the software must actually cease broadcasting during idle times while still listening to detect wake-up bursts.

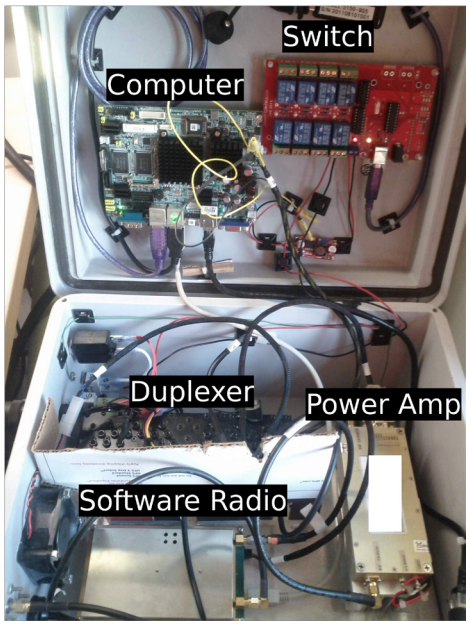


Figure 1: Our Range Networks GSM BTS.

Hardware Figure 1 shows the internals of our revised Range Networks 5150 cellular base station. The key pieces of equipment are the radio, computer, duplexer, and power amplifier (PA). We added a USB-controlled high-current switch and connected it directly to the power amplifier, allowing us to control the PA's status via serial commands from the computer. When the BTS enters idle mode, the PA is turned off.

Software There are two key software modifications. First, we implement the idle mode and drop all transmissions (including the beacon) while the power amplifier is off. Second, we implement a mechanism for the BTS to receive wake-up signals from user radios.

We implemented idle mode with a service that sends messages to the switch controlling the power amplifier. This daemon, which has access to the GSM and switch state, controls entering and exiting idle mode. Instead of naively returning to idle when all calls have terminated, we use a number of heuristics to improve the user experience. First, we require that the network be active for a minimum of 90 seconds, approximately double what we found to be the worst-case time necessary for a handset to connect to and communicate with the tower (i.e., *camping*) (Table 3). This ensures that all handsets waiting to camp will have ample time to do so should a tower return from idle mode. Second, the BTS only transitions to idle if there has been no cellular traffic for 30 seconds. This enables serialized actions, e.g., redialing a dropped call.

Originally, we had hoped to provide a "low coverage mode" (i.e., signal transmission without amplification),

where the BTS could still provide coverage for people physically near the radio. In testing, however, we discovered that the radio *must* be disabled when in idle, even to the exclusion of transmitting with the amplifier in pass-through mode. If the BTS broadcasts (even at low power), handsets nearby will attempt to camp (a process near-all handsets perform automatically and periodically). As our burst-detection is a simple power measurement, this legitimate network traffic would be indistinguishable from wake-up bursts.

The BTS provides two primary functions: incoming (mobile-terminated) and outgoing (mobile-originated) calls. Mobile-originated calls are simple. The tower must be in active mode, as only a camped handset can initiate a call. Mobile-terminated calls are more complicated. If the tower is "active" when it receives a call, the call is simply routed to the appropriate handset. If the tower is "idle", the caller either leaves a voicemail (and the tower remains "idle") or they are put on hold and the tower immediately wakes and waits for the handset to camp. According to our measurements (Table 3), this can be up to 40 seconds (with most being under 25 seconds). When the handset eventually camps, the callee is immediately connected to the caller by bridging to the held call or initiating a new call if they hung up.

The basic mechanism for detecting bursts is implemented in the transceiver of the radio. If the radio is in idle mode, any high enough power burst on the tower's Absolute Radio Frequency Channel Number (ARFCN, basically just the frequency the tower listens and transmits on) will cause a message to be sent to the daemon, waking the system. The power required depends on the current noise level as determined by the transceiver. This technique is similar to ones used in sensor networks [13].

As OpenBTS utilizes voice-over-IP (VoIP) as it's interconnect, there are no changes required to any other network services. Were we to interconnect using more traditional protocols (i.e., SS7/MAP) the name database (HLR) would have to allow longer registrations from users on virtual coverage enabled BTSs. This is the only change required for inter-operation.

With this system, we are able to provide on-demand voice services for rural networks at the cost of increased call-connection latency. SMS and data traffic are assumed to sync during periods of active voice traffic.

4.2 Waking Up in Virtual Coverage

Virtual coverage is not just a change in the cellular tower; it also requires a device capable of sending a "wake up" message. As mentioned in the previous section, we implemented two mechanisms for sending this message: from a handset or via an autonomous radio.

4.2.1 Cellular Handset

We have implemented our base station wake-up mechanism using an osmocomBB compatible mobile phone. We call this the *Wake-up Phone (WUP)*. OsmocomBB [24] is an open-source GSM baseband software implementation, which simplifies changes to the GSM protocol. However, every GSM handset should be able to send a wake-up burst with a software change from the manufacturer. The mechanism for waking up the BTS is sending a burst packet on the BTS's ARFCN. The BTS, though not transmitting, receives this message and exits the idle state, allowing the handset to camp.

Each BTS broadcasts its ARFCN number (as well as the ARFCNs of similar nearby towers) on the beacon channel, which details the exact frequency used to communicate with the BTS. A handset periodically scans the network for towers to camp on, and gathers these numbers. In our system, the handset stores these numbers when the network idles, and then uses them to send "wake up" messages (as above) during periods without network availability.

Mobile-Originated (MO) Call In order to initiate a call, the Wake-up Phone will transmit "wake up" bursts on a selected set of ARFCNs. These ARFCNs are either a list of previously detected base stations or a static configured list. The "wake up" packets are random packets that are transmitted on the selected ARFCN. After transmission, handset scans for a tower broadcasting on the ARFCN just awoken, instead of scanning the whole cellular band (as in normal cell selection). If discovered, the handset camps to this tower and the user is able to communicate.

If a WUP is unsuccessful in camping to the recently awakened base station, the handset will proceed to the next ARFCN in its list, if any, and perform similar operations. This mechanism repeats until the handset is successfully camped or it runs out of available ARFCNs. At this point it will default back to the standard GSM protocol, which scans the entire band looking for available towers.

Mobile-Terminated (MT) Call As stated above, when the BTS receives a mobile-terminated call it immediately exits idle mode and waits for the handset to camp. The WUP scans the stored ARFCNs much more frequently (10:1), reducing the average time to camp. However, this does not affect the worst case analysis, which is 7s. When found, the phone camps and the call is connected.

4.2.2 Wake-up Radio

We have also designed and implemented a system to wake-up our BTS, the *Wake-up Radio (WUR)*,

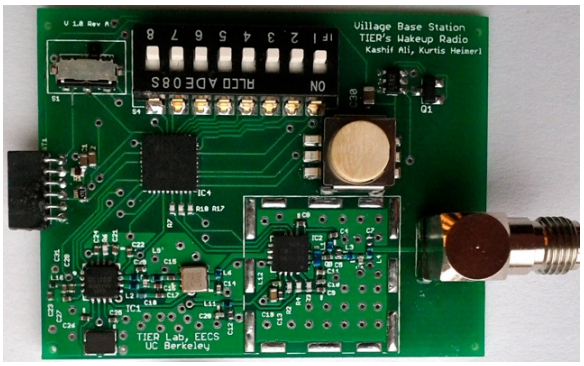


Figure 2: Prototype implementation of WUR.

Component	Draw (W)	% (10W)	% (50W)
Computer	12W	17.4%	7.8%
Radio	12W	17.4%	7.8%
10W Amp	45W	65.2%	
50W Amp	130W		84.4%

Table 2: Power draw of the components of our Range Networks 5150 BTS.

nicknamed the *garage door opener*. The WUR transmits wake-up bursts, similar to our modified GSM handset, on a specific ARFCN.

The radio is designed to be both cheap and low power. The primary user interface is just a single button. When pressed, this button triggers a burst on the configured ARFCN. The radio produces a signal at approximately 500mW, the minimum power required for a handset in the GSM standard. The WUR uses an on-board battery pack, but provides interfaces for other power sources (e.g., solar) as well. The WUR, with two AAA batteries is capable of 5000+ bursts. The WUR can be configured to produce different ARFCNs with dip switches.

The WUR is only needed for mobile-originated communications. For mobile-terminated communications, the tower simply wakes and waits for the recipient's handset to camp.

5 Technical Evaluation

5.1 BTS Power Savings

We begin by evaluating the performance of a single modified Range Networks 5150 BTS. This unit has two power amplifiers available: 10 Watt and 50 Watt. The 10W unit supports just one channel and seven concurrent calls. It is commonly used for low-density areas. The 50W unit is designed for denser areas, produces up to five channels, and is capable of providing 35 concurrent calls. Both towers cover up to 35 kilometers, depending on configuration and geography. Areas with buildings or dense foliage will have worse signal propagation.

Model	Time (Avg)	Time (Max)
WUP (2G) (MT)	2s	7s
WUP (2G) (MO)	2s	2s
HTC Dream (3G)	12.1s	41.8s
Samsung Nexus S (3G)	23.6s	37.6s
Nokia 1202 (2G)	10.8s	14.6s

Table 3: Measurements on how long a handset has to wait, on average, to camp to a specific tower. This is the additional connect time if the network is idle.

Table 2 shows the relative power draw for each component of the BTS. As expected, the power amplifier dominates the overall power draw, consuming 65% of the power for a 10W unit and 84% for a 50W.

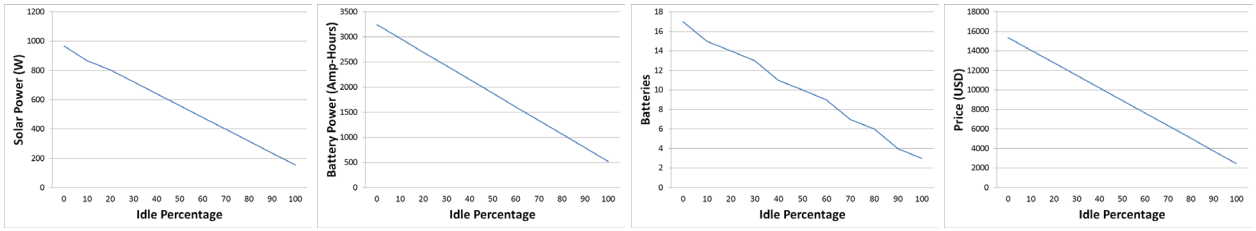
In our system, we added a USB-controlled switch to programmatically control the power amplifier. This switch draws negligible power (less than 1W). We also saw no change in the power draw of the computer, as expected with the BTS handling no calls in "idle" mode. As such, we are able to reduce the overall power draw of our BTS by over 65%.

5.2 Handsets

Wake-up Radio With the wake-up radio (WUR), the user has access to a device tuned to the particular frequency of their local cellular tower. Depending on local cost and logistic constraints, this device can be either widely deployed as an attachment to each local user's individual cellular phone, or singly deployed at some central location as a "phone booth". The user presses the button, sending the wake-up message to the tower, taking the station out of idle mode. When the tower wakes, it broadcasts a beacon signaling its location and ownership.

Traditional GSM handsets periodically scan the airwaves looking for beacons. As such, a user's handset will eventually camp to the newly awoken BTS. We measured the time to camp, after waiting to ensure the initial network search failed, for three different handsets: the Samsung Nexus S (Android), the HTC Dream (Android), and the Nokia 1202 (Symbian S30) over thirteen trials. The results are shown in Table 3. The first two phones are quad-band 3G phones, meaning that they scan a wider band than a dual-band 2G-only feature phone (e.g., our Nokia 1202) commonly used in developing regions.

Users must wait for their handsets to camp in order to communicate using the network. Our results mean that using the WUR increases the setup time for all calls by a maximum of approximately 40 seconds. The average wait measured is less than 25 seconds. Both the maximum and average time to camp are highly dependent on the specific phone used. Though this



(a) (b) (c) (d)

Figure 3: Solar Power (a), Battery Power (b), Individual Batteries (c), and total spending (d) required to operate a virtual coverage tower at a certain amount of idleness for a week in an area with 5 hours of sun. Note that 0% idle is equivalent to a traditional tower.

potential wait is a nontrivial amount of time, we believe this is acceptable to rural users who have limited alternatives for communication.

Wake-up Phone In the embedded solution, the WUP is able to camp on the BTS almost immediately after sending the wake-up burst. Since the same hardware device both delivers the burst and camps to the tower, the timing of the two tasks can be optimized for minimal delay. We measured the amount of time needed to register with the BTS [1] and found that, in all cases, it took exactly two seconds from wake-up burst to "camped normally". This is shown in Table 3. The practical impact of this result is that, with a virtual coverage network, every mobile-originated communication takes two seconds longer to set up when the BTS starts idle.

For mobile-terminated calls, the handset does not generate the wake-up burst and must instead listen for a cellular beacon. Fortunately, the handset still knows what cellular towers are in the area. Instead of periodically scanning the entire band (as in standard GSM), we can scan just the beacons that are present in the area. This again takes no more than two seconds. Other operating system functions occasionally delay this scan, causing a maximum wait time of seven seconds.

5.3 Deployment Example

To begin to understand the impact of virtual coverage in a real-world situation, we calculate the approximate amount of infrastructure (solar panels and batteries) required to support a 50W (drawing 155W) cellular station year-round using only solar power.

We first frame our "real-world situation": Providing winter-time network coverage in the South Asian country profiled in our later evaluation. During this time period, the country receives 5 hours of usable sun [10], as solar panels deliver minimal power when the sun is on the horizon. Using this, we are able to calculate the amount of power optimally tiled solar panels generate. We assume an operating temperature of 40 degrees Fahrenheit and 24V batteries. Batteries are priced at 442 USD for 200 Amp-Hours and solar panels are priced at 1.07 USD per Watt. Lastly, the BTS draws 155W at full power (3720 Watt-hours/day) and 25W at idle (600 Watt-hours/day). As there is often inclement weather, we calculate the requirements for powering the station over a week without any power generation. The results are shown in figure 3.

The actual impact of virtual coverage is large; a completely idle tower requires one sixth of the batteries, solar panels, and total infrastructure cost of a traditional tower. As expected, these variables scale linearly with increasing idleness. We later (Section 6.3) show that idleness scales sub-linearly with respect to total calls (and thus users), meaning that the price of infrastructure required to support a virtual coverage tower scales sub-linearly with the total number of calls handled. Contrast this with a traditional cellular tower that must install the same amount of solar panels and batteries regardless of the number of calls and users serviced.

We wish to note that these costs are not only monetary. A single traditional BTS wanting week-long backup requires a kilowatt of solar panels and seventeen deep-cycle batteries (each weighing 68 pounds!). This equipment will be hiked into rural areas, an enormous load. Compare that with a virtual coverage station in an 80% idle area. There, just 300W of solar panels and 6 batteries must go up the hill. Lastly, virtual coverage also allows for growth; as an area moves from 80% to 70% idleness, new batteries and panels can be installed.

6 Real-World Evaluation

In addition to the micro-benchmarks just presented, it is important to understand how our system would perform *in situ*. For although we've demonstrated that our modified base station saves power and provides a consistent user experience, it requires periods of idle time in individual base stations, and this important variable is not known.

[1] GSM state *A1_TRYING_RPLMN* to *C3_CAMPED_NORMALLY*

To resolve this, we use traces from existing cellular networks to provide a "worst case" analysis of virtual coverage's benefits to areas without coverage. Specifically, we run two simulations using log data obtained from two mobile telecommunications firms in developing countries. Our simulations calculate the amount of power that the operator would save if they implemented our system of virtual coverage, based on actual patterns of idleness. Although these networks service millions of users (and not the rural areas we target), we show that our technology has the potential to save substantial amounts of power by utilizing the available idle time. [2]

6.1 Data

To assist in evaluating our research question, we acquired data from two large mobile telecommunications operators, one in sub-Saharan Africa and one in South Asia. The data we utilize contains a detailed log of tower activity over the span of one week, including information on: the start time of each mobile-originated and mobile-terminated call, the duration of the call (in seconds), and the approximate location of each tower. Note that all cellular activity is *per-tower* and not *per-cell*. We do not know how many cells were located on each tower; a negative bias as our system is capable of idling per-cell.

Although we are contractually bound not to disclose the identity of the mobile operators whose data we analyze, some basic facts are relevant.

Sub-Saharan Africa (SSA) The first operator's data comes from a sub-Saharan African nation. Towers in this country are heavily utilized, although overall mobile phone penetration during the week we analyze was roughly 20 percent. We observe 15 million calls from a representative random sample of approximately 150 unique towers. There are an average of 90000 calls per tower, with a median of near 70000.

South-Asia (SA) The second operator's data comes from a large country in South Asia. Mobile penetration here was roughly 55 percent during the week we analyze, but average tower utilization was lower than SSA. We observe 35 million calls from a representative random sample of around 5000 unique towers. There are an average of 6000 calls per tower, with a median of 4000.

[2]It is worth noting that we do not advocate replacing existing telecommunications networks with our equipment. As stated above, we are designing for autonomous rural networks. However, we believe it is an instructive demonstration of the value of virtual coverage.

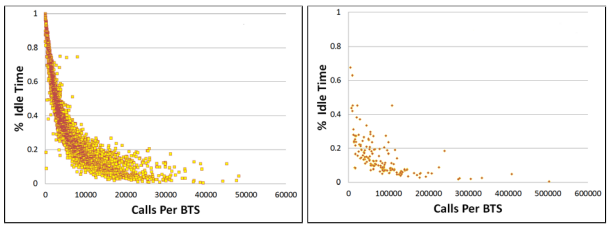
6.2 Analysis

Combining the call log data with our user experience extensions (Section 4.1), we determine the amount of time where the network can be put into an "idle" mode while still providing complete cellular service to all users. We separately compute results during daytime (6am-6pm) and nighttime (6pm-6am), as user behavior and power generation are different during these periods.

We begin by determining the relative amount of idleness in the cellular network by using the detailed logs to determine when users initiate actions on the network. Of course, the network cannot power down for every idle period; users must have time to camp on the network and they may wish to communicate multiple times without having to wake the tower up repeatedly. With this in mind, we instead model a realistic user experience of the network. In this model, we assume the tower will remain awake for some time after a logged call in case a new call is placed or received. In our network, the towers remain active for 30 seconds following any user-initiated action. Each tower must also stay available for a minimum of 90 seconds at a time; this ensures that any phones within virtual coverage range waiting to initiate an action have enough time to detect and connect to the network (noting that the prior section found a maximum of 42 seconds to camp when scanning in disconnected mode). Any idle periods seen in the logs greater than 24 hours are removed from consideration (rather than being "idle"), as they are almost certainly a power outage. We found 18 such periods in SSA (in 25000 hours of coverage) and 59 such periods in SA (in 940000 hours).

6.3 Results: Idleness

Figure 4 shows the relative amount of idleness vs total number of calls per base station in both sub-Saharan Africa and South Asia. We note that the amount of idle time in the network (rural or urban) scales *sub-linearly* with the total number of calls. Fitting a logarithmic trend line (occupied time vs number of calls) results in $y = -0.077ln(x) + 1$ for SSA, and $y = -2.04ln(x) + 2.8$ for SA. This result means that each new call causes less occupied network time than the last, on average. This makes sense; as the number of calls on a tower increases, the amount of overlapping network activity also increases. Overlapping calls have zero marginal cost (in terms of power), and so we see this sub-linear benefit. This result, combined with our previous result showing that the cost of power infrastructure scales linearly with the idle percentage (Section 5.3) means that virtual coverage allows infrastructure cost to scale sub-linearly with the total number of calls serviced, and presumably the number of users serviced. As a byproduct of this



(a) (b)

Figure 4: The idle time for both South Asia (a) and sub-Saharan Africa (b).

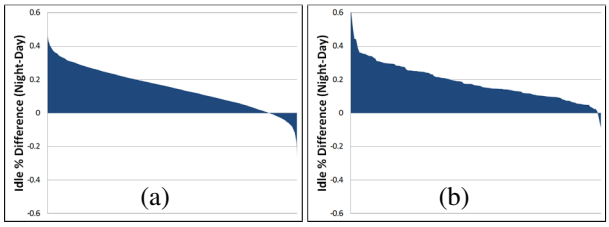


Figure 5: The difference in day and night idle time for each of the cellular towers in SA (a) and SSA (b), sorted by difference.

result; rural entrepreneurs can invest in more power infrastructure as demand grows, rather than requiring a large capital expenditure during installation.

Figure 5 next compares the day and night idle time in the network. This comparison is done for each BTS in the study, and over five thousand towers are difficult to represent graphically. To resolve this, we instead graph the *difference* between the night and day idle time. For instance, a tower that is idle 25% at night and 5% during the day would be represented by a point at (.25-.05)=.2.

For sub-Saharan Africa, we see more idle time at night. Over 98% of the towers are more idle at night than during the day. This result is diminished in South Asia, with just under 89% of the towers being more idle at night than during the day. We believe this is primarily due to a differential pricing scheme used in SA to encourage nighttime communications.

These results suggest that we should target night idle periods for saving power. This also works well with solar, since night power requires storage and thus costs considerably more due to both lower efficiency (10–20% loss), and the ongoing cost of battery replacement.

Lastly, our data also demonstrates the enormous amount of idle time available in these networks at night. 86% of the towers in SA are over 20% idle at night, while 53% of those in SSA pass the same metric. There is a significant opportunity for virtual coverage to reduce power consumption expenditures in both networks.

6.4 Results: Power Savings

We begin by noting the power measurements in Section 5.1. These measurements indicate that high-

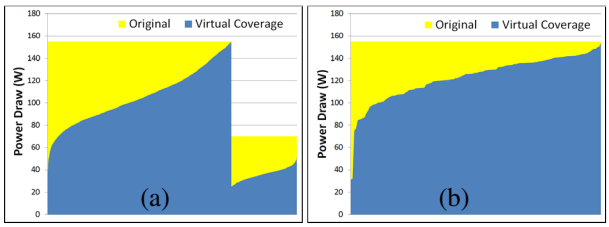


Figure 6: Comparison of the power saved by every BTS in SA (a) and SSA (b) by using virtual coverage.

	Power Draw	Savings %
SA Original	1483 kW	**0%**
SA Day	584 kW	**21.3%**
SA Night	488 kW	**34.3%**
SA Total	1071 kW	**27.7%**

Table 4: The final results of our network simulation in South Asia. There are always significant power savings.

capacity towers (greater than 7 concurrent calls) require larger amplifiers, drawing 155W at full power and 25W at idle (a savings of 84%). Smaller capacity towers (supporting less than 7 concurrent calls) draw 70W at full power and 25 at idle (a savings of 65%). We use these measurements, combined with the measures of the maximum number of concurrent calls observed on each tower, to estimate the power draw of each individual BTS on both networks.

Figure 6 shows the results of this calculation. Each tower's original power draw (yellow) is compared directly against the same tower using virtual coverage (blue). There is significant power savings in each network. We now move to calculate the exact amount of power saved.

Tables 4 and 5 show the final results if we sum the power drawn (and saved) by all tower equipment observed. Using virtual coverage, we are able to reduce the total night power budget by 34% for our South Asian network, and 21% in sub-Saharan Africa. During the day, when solar power is more available, we found that the network power draw could be reduced by 21% in South Asia, and just 7% in sub-Saharan Africa.

We wish to remind the readers that this analysis is on two *existing* cellular networks with broad coverage

	Power Draw	Savings %
SSA Original	45.6 kW	**0%**
SSA Day	21 kW	**7.2%**
SSA Night	18 kW	**20.7%**
SSA Total	39 kW	**13.9%**

Table 5: The final results of our network simulation of the sub-Saharan Africa network. Significantly more power is saved at night.

on nation-wide scales. Our system is designed to provide a mechanism for covering parts of the world currently without coverage, and gathering call records from such a place is impossible. We hope this proxy measurement, demonstrating massive possible power savings in networks that are economically feasible is a suitable demonstration that there would be similar (likely better) savings in areas currently without network.

These power savings (34% SA, 21% SSA), when combined with our earlier technical evaluation showing limited impact on the user experience (adding an average of 25 seconds to each call) and sublinear scaling of power infrastructure cost with regards to total calls, demonstrate a compelling system. It is low-cost, low-power, and well-suited for wide-area cellular communications in off-grid areas dependent on renewable energy sources.

7 Discussion

Data/SMS Services An obvious critique of our work is that we do not actively support SMS or data services on the base stations. In particular, as presented we do not wake up base stations for those messages alone.

SMS as a protocol already incorporates delay-handling, and functions asynchronously with respect to sender and recipient. There exist opportunities to improve SMS user experience in a virtual coverage cell. For instance, the carrier could set a maximum message delay: if a tower receives a message for one of its handsets when idle, the tower must activate and deliver the message within one hour. This has the potential to limited wasted active network time and the carrier is free to establish this interval based on their own preferences.

If a user requests data service while the tower is idle, the BTS can handle this in a similar fashion to placing a phone call: the tower is sent a wake-up burst (via either phone handset or wake-up radio). If idle time is significantly reduced by data requests (e.g., by data-channel apps seeking updates from the web), users can be incentivized to turn off these features when negotiating service rates with the provider.

Mobility As stated in Section 2, we explicitly avoid the issue of mobility in this paper. Our equipment is designed to create "islands" of coverage, simplifying the architecture dramatically. It is assumed that a rural virtual coverage cell will not intersect any other covered region. However, as this work moves forward, we recognize that the issue of mobility should be addressed.

The GSM tower broadcasts not only its own ID, but also those of nearby towers. This helps in two ways. First, a handset can try waking up towers in succession to increase its chances of successfully waking a BTS, at the cost of extra delay. Second, during a call the handset could try to wake-up nearby towers proactively, either due to low signal from the current tower or just in case. Once awake hand-off to neighboring towers works as usual. Finally, on higher-end phones, a GPS-indexed database could inform the handset of exactly what tower(s) to wake in a specific location.

Inter-operation with existing infrastructure The GSM specification assumes constant coverage; a by-product of a system designed for developed, urban areas with strong power infrastructure. Virtual coverage changes this, turning cellular towers into dynamic agents. The interaction of these networks is complex.

Our system handles this already: Modified handsets *always* connect and call through existing static systems if possible: we are only capable of waking a tower if we are not attached to any existing tower. This is done primarily to save power; the static tower is on (whether or not it fields an additional call), but we'd prefer to keep our local tower idle. Similarly, if the other tower is dynamic but active, we'd prefer to send two calls through the powered tower, rather than waking an idle tower.

Security The system, as designed, has no mechanism for authenticating users before they wake the BTS. This means that one dedicated attacker could launch a power-based denial of service (DOS) attack by constantly sending the wake-up bursts to the tower. For example, a dedicated user may DOS the tower to leverage an information asymmetry they have developed.

The two mechanisms for waking up the BTS have different trade-offs for preventing such DOS attacks. For the WUR it is impossible to identify the user sending the wake-up burst, as the device is totally separate from the phone. Instead, we could add a cost to the use of the physical radio; perhaps by placing it in a phone-booth-like structure and charging a fee. This way, DOSing the tower could be made prohibitively expensive.

For the modified handset, we can identify the users by changing the protocol slightly. Currently, users send the burst, camp, and then wait. They must choose separately to make a call. We can instead enforce that the user must make a call (or any communication) immediately after camping to the station. This would allow us to know who did the waking. Unfortunately, this is still susceptible to another, similar attack; a user could send the wake-up burst and immediately pull the battery. In this case, we could modify the BTS to note there has been no traffic and immediately re-enter the idle state, reducing the impact of the attack.

Lastly, it is possible to send identifying information, such as the IMSI (unique SIM ID), in the burst message. This would allow us to charge users for waking the tower and prevent DOS attacks. We leave this to future work.

8 Related Work

Virtual coverage utilizes ideas first developed for shorter-range wireless sensor networks to save power in wide-area cellular networks. It also makes use of the recent advances in open-source telephony to implement these power-saving techniques. Lastly, our focus is on the problem of rural connectivity.

8.1 Saving Power

As the world goes "green", reducing power consumption has become a critical goal of system designers [4]. Researchers (and system implementers) have focused on many different mechanisms for saving power, including disabling specific pieces of hardware. Gobriel et al. [11] investigated how to save power by utilizing the "idle time" in data center networks. Zheng et al. [34] used asynchronous wake-up mechanisms to save power in ad-hoc wireless networks. Wake-on-Wireless-LAN [21, 30] enables behavior similar to wake-on-LAN [22] across wireless networks. We similarly save power by utilizing idle time and a wake-up mechanism in cellular networks. However, the potential power savings in rural GSM cellular networks far outweigh smaller wireless deployments, as the range and power consumption of cellular networks are orders of magnitude larger.

8.2 Sensor Networks

Efficient use of power is of critical importance in wireless sensor networks due to battery-based operation. The core technique we use to save power in cellular networks is very similar to one developed by Gu et al. [13] for wireless sensor networks. Other researchers have explored similar designs [7]. As in our system, the nodes sit idle until communication is needed, and wake each other up using large radio bursts that are distinguishable from noise through some mechanism. Similarly, other researchers also created standalone devices for creating wake-up signals [8]. This is unsurprising, as these technologies inspired our design. The key differences are in scale, intent, and mechanisms.

8.3 GSM/Cellular

The OpenBTS project [23] is a full-featured GSM base station using software-defined radios. It bridges GSM handsets to voice-over-IP systems, enabling cheaper, lower-power cellular equipment. The OsmocomBB project [24] is an open-source GSM handset, capable of interfacing with any 2G GSM station. Our research is built on these two pieces of technology that enable us to modify the GSM standard and implement a low-power GSM telephony solution.

Lin et al. investigated multihop cellular networks [20] to broaden the range of GSM cellular towers. In their solution, each handset could potentially act as a router for other handset's messages, directing them to a centralized base station. Others expanded this idea, investigating how one might property incentivize users to share their cellular connections [19]. Our work utilizes a different mechanism, virtual coverage, to achieve the goal of increased effective network range. These two designs are not mutually exclusive. A solution using both could have a dramatic impact on rural telephony.

A few groups have explicitly investigated reducing power consumption in cellular networks. Bhaumik et al. [5] proposed varying the size of cells; saving power by turning a subset of the cellular towers off during times of low load. Peng et al. [27] took these ideas even farther, demonstrating significant over-provisioning in cellular networks. Unfortunately, their proposed method for saving power is not possible in most rural areas as there is often just one tower providing service.

8.4 Developing Regions/Rural Networks

Network connectivity in developing and rural areas is an area of active research [6]. Wireless telecommunications are a common idiom [31], as the cost of deploying these networks is significantly less than traditional wired networks in areas with limited infrastructure. Researchers have investigated long-distance wireless [26] and sustainability in these areas [32].

Seeing the need for rural cellular communications, commercial providers have begun to develop products optimized for these areas. Both Vanu [33] and Range Networks [28] have developed "low power" 2G cellular equipment capable of running off of entirely renewable energy and using only around 90 watts of power. Altobridge's [2] lite-site product varies its capacity in order to save power. As all of these products provide constant network coverage (rather than virtual coverage), they cannot reduce their power draw below 90 watts.

The rural/urban divide is also an area of active work. Eagle et al. [9] investigated how calling patterns changed as users migrated between rural and urban areas. Heimerl et al. [14, 15, 16] researched how users in developed, developing, urban, and rural areas viewed and made use of their cellular infrastructure. He found that rural users had a better understanding of network properties, including coverage patterns. This work informs our decision to include users in network power provisioning, as we expect them to quickly understand and make use of the basic primitives provided by our system.

Figure 7: Installing the Village Base Station in rural Papua, Indonesia.

9 Future Work

We're currently deploying the Village Base Station [14], complete with virtual coverage, in rural Papua, Indonesia (Figure 7). This is a longitudinal study investigating not only the impact of virtual coverage, but also the general feasibility of off-grid, off-network, rural, community-owned and supported cellular installations. We hope to have this work completed by mid 2013.

10 Conclusion

The positive impact of mobile phones on the poor is one of the great ongoing success stories of development, with more users and more impact than any other advanced technology. The arrival of low-cost devices, and even low-cost smart phones, make mobile phones the best platform for current and future interventions, including mobile banking, education, health care, and governance. Yet much of the rural world lacks coverage due to the high cost of infrastructure in rural areas; reaching the next billion+ users will be much harder now that most urban areas have coverage.

In this work we presented virtual coverage, a mechanism for dramatically reducing the power draw of cellular infrastructure in rural areas. This is done by introducing an "idle" mode to the network, similar to work done in wireless sensor networks. Instead of providing constant coverage (wasted in times of low communication, such as at night), we provide coverage only when needed. A user demonstrates their need to communicate with one of two mechanisms. First, we modified the baseband of a cellular phone to send a "wake up" message when a user wants coverage. Second, recognizing that modifying the baseband of the billions of cellular phones already deployed is likely infeasible, we developed a custom low-cost radio capable of producing the same signal. These changes utilize existing manufacturing economies, requiring just a small hardware modification to the BTS, a software change to the handset, and the manufacturing of a $14 device.

We validated this design by implementing the system and demonstrating the power savings. We showed that, with proper use, our equipment saves between 65%-84% of the power at idle. We measured the impact on users, who would see an average of less than 25 seconds added to any call. Users of our custom firmware would see just two seconds delay. We showed that a virtual coverage installation could be built with one-sixth of the power infrastructure of a traditional tower. We demonstrated that the power requirements for a virtual coverage tower scale sub-linearly with the total number of calls (and presumably callers) serviced. This allows smaller operators to invest in their network as it grows, rather than having the entire expenditure be up front.

We also simulated both an sub-Saharan African and South Asian cellular carrier using our system. We found that we are able to save 34% of the night power (21% during the day) in South Asia. For the denser sub-Saharan African country, we can save 21% of the power at night and 7% during the day. This reduction in power consumption enables more use of solar power and makes cellular system more economically viable in rural areas far from grid power or network.

11 Code

All of our software and hardware designs are open source. Our modified versions of OpenBTS, OsmocomBB, and the schematics for the Wake-up Radio are in the following repositories:

- https://github.com/kheimerl/openbts-vbts
- https://github.com/kheimerl/osmocom-bb-vbts
- https://github.com/kheimerl/VBTS

All of our systems can be run on open hardware, primarily the Ettus USRP line of products.

12 Acknowledgments

Thanks to Tapan Parikh, Kelly Buchanan, Stephen Dawson-Haggerty, Andrew Krioukov, Nathan Eagle and Amy Wesolowski.

References

[1] Unscheduled loadshedding irks people in punjab. *The Nation (Pakistan)*, October 2011.

[2] Altobridge. http://www.altobridge.com/. Retrieved 8/2012.

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[4] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, Dec. 2007.

[5] S. Bhaumik, G. Narlikar, S. Chattopadhyay, and S. Kanugovi. Breathe to stay cool: adjusting cell sizes to reduce energy consumption. In *Proceedings of the first ACM SIGCOMM workshop on Green networking*, Green Networking '10, pages 41–46, New York, NY, USA, 2010. ACM.

[6] E. Brewer, M. Demmer, B. Du, M. Ho, M. Kam, S. Nedevschi, J. Pal, R. Patra, S. Surana, and K. Fall. The case for technology in developing regions. *Computer*, 38(6):25–38, 2005.

[7] I. Demirkol, C. Ersoy, and E. Onur. Wake-up receivers for wireless sensor networks: benefits and challenges. *Wireless Commun.*, 16(4):88–96, Aug. 2009.

[8] B. V. d. Doorn, W. Kavelaars, and K. Langendoen. A prototype low-cost wakeup radio for the 868 mhz band. *Int. J. Sen. Netw.*, 5(1):22–32, Feb. 2009.

[9] N. Eagle, Y.-A. de Montjoye, and L. M. A. Bettencourt. Community computing: Comparisons between rural and urban societies using mobile phone data. In *Proceedings of the 2009 International Conference on Computational Science and Engineering - Volume 04*, CSE '09, pages 144–150, Washington, DC, USA, 2009. IEEE Computer Society.

[10] Global Energy Network Institute. https://www.geni.org/. Retrieved 8/2012.

[11] S. Gobriel, C. Maciocco, and T.-Y. Tai. Long idle: Making idle networks quiet for platform energy-efficiency. In *Systems and Networks Communications (ICSNC), 2010 Fifth International Conference on*, pages 340 –347, aug. 2010.

[12] GSM Association. Powering Telecoms: East Africa Market Analysis Sizing the Potential for Green Telecoms in Kenya, Tanzania and Uganda. http://www.gsma.com/mobilefordevelopment/powering-telecoms-east-africa-market-analysis. Retrieved 2/2013.

[13] L. Gu and J. A. Stankovic. Radio-triggered wake-up for wireless sensor networks. *Real-Time Syst.*, 29(2-3):157–182, Mar. 2005.

[14] K. Heimerl and E. Brewer. The village base station. In *Proceedings of the 4th ACM Workshop on Networked Systems for Developing Regions*, NSDR '10, pages 14:1–14:2, New York, NY, USA, 2010. ACM.

[15] K. Heimerl, R. Honicky, E. Brewer, and T. Parikh. Message phone: A user study and analysis of asynchronous messaging in rural uganda. In *SOSP Workshop on Networked Systems for Developing Regions (NSDR)*, 2009.

[16] K. Heimerl and T. S. Parikh. How users understand cellular infrastructure. Technical Report UCB/EECS-2012-42, EECS Department, University of California, Berkeley, Apr 2012.

[17] International Telecommunications Union. Green Solutions to Power Problems (Solar & Solar-Wind Hybrid Systems) for Telecom Infrastructure. www.itu-apt.org/gtas11/green-solutions.pdf. Retrieved 1/2013.

[18] International Telecommunications Union. The World in 2010: ICT Facts and Figures. http://www.itu.int/ITU-D/ict/material/FactsFigures2010.pdf. Retrieved 2/2011.

[19] M. Jakobsson, J.-P. Hubaux, and L. Buttyn. A micro-payment scheme encouraging collaboration in multi-hop cellular networks. In W. Hunt and F. Somenzi, editors, *Computer Aided Verification*, volume 2742 of *Lecture Notes in Computer Science*, pages 15–33. Springer Berlin / Heidelberg, 2003.

[20] Y.-D. Lin and Y.-C. Hsu. Multihop cellular: a new architecture for wireless communications. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 3, 2000.

[21] N. Mishra, K. Chebrolu, B. Raman, and A. Pathak. Wake-on-wlan. In *Proceedings of the 15th international conference on World Wide Web*,

WWW '06, pages 761–769, New York, NY, USA, 2006. ACM.

[22] S. Nedevschi, J. Chandrashekar, J. Liu, B. Nordman, S. Ratnasamy, and N. Taft. Skilled in the art of being idle: reducing energy waste in networked systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 381–394, Berkeley, CA, USA, 2009. USENIX Association.

[23] OpenBTS. http://openbts.sourceforge.net/. Retrieved 4/2012.

[24] OsmocomBB. http://bb.osmocom.org/trac/. Retrieved 4/2012.

[25] PapuaCom, July 2012. Personal Communication.

[26] R. Patra, S. Nedevschi, S. Surana, A. Sheth, L. Subramanian, and E. Brewer. Wildnet: Design and implementation of high performance wifi based long distance networks. In *4th USENIX Symposium on Networked Systems Design and Implementation*, 2007.

[27] C. Peng, S.-B. Lee, S. Lu, H. Luo, and H. Li. Traffic-driven power saving in operational 3g cellular networks. In *Proceedings of the 17th annual international conference on Mobile computing and networking*, MobiCom '11, pages 121–132, New York, NY, USA, 2011. ACM.

[28] Range Networks. http://www.rangenetworks.com/. Retrieved 8/2012.

[29] Range Networks, September 2011. Personal Communication.

[30] E. Shih, P. Bahl, and M. J. Sinclair. Wake on wireless: an event driven energy saving strategy for battery operated devices. In *Proceedings of the 8th annual international conference on Mobile computing and networking*, MobiCom '02, pages 160–171, New York, NY, USA, 2002. ACM.

[31] L. Subramanian, S. Surana, R. Patra, S. Nedevschi, M. Ho, E. Brewer, and A. Sheth. Rethinking wireless in the developing world. In *In Proc. of the 5th Workshop on Hot Topics in Networks (HotNets*, 2006.

[32] S. Surana, R. Patra, S. Nedevschi, and E. Brewer. Deploying a rural wireless telemedicine system: Experiences in sustainability. *Computer*, 41(6):48–56, 2008.

[33] Vanu. http://www.vanu.com/index.html. Retrieved 8/2012.

[34] R. Zheng, J. C. Hou, and L. Sha. Asynchronous wakeup for ad hoc networks. In *Proceedings of the 4th ACM international symposium on Mobile ad hoc networking & computing*, MobiHoc '03, pages 35–45, New York, NY, USA, 2003. ACM.

EyeQ: Practical Network Performance Isolation at the Edge

Vimalkumar Jeyakumar[1], Mohammad Alizadeh[1,2], David Mazières[1], Balaji Prabhakar[1], Changhoon Kim[3], and Albert Greenberg[3]

[1]*Stanford University* [2]*Insieme Networks* [3]*Windows Azure*

Abstract

The datacenter network is shared among untrusted tenants in a public cloud, and hundreds of services in a private cloud. Today we lack fine-grained control over network bandwidth partitioning across tenants. In this paper we present EyeQ, a simple and practical system that provides tenants with bandwidth guarantees as if their endpoints were connected to a dedicated switch. To realize this goal, EyeQ leverages the high bisection bandwidth in a datacenter fabric and enforces admission control on traffic, regardless of the tenant transport protocol. We show that this pushes bandwidth contention to the network's edge, enabling EyeQ to support end-to-end minimum bandwidth guarantees to tenant endpoints in a simple and scalable manner at the servers. EyeQ requires no changes to applications and is deployable with support from the network available today. We evaluate EyeQ with an efficient software implementation at 10Gb/s speeds using unmodified applications and adversarial traffic patterns. Our evaluation demonstrates EyeQ's promise of predictable network performance isolation. For instance, even with an adversarial tenant with bursty UDP traffic, EyeQ is able to maintain the 99.9th percentile latency for a collocated memcached application close to that of a dedicated deployment.

1 Introduction

In the datacenter, we seek to virtualize the network for its tenants, as has been done for compute and storage. Ideally, a tenant running on shared physical infrastructure should see the same range of control- and data-path capabilities on its virtual network, as it would see on a dedicated physical network. This vision has been in full swing for some years in the control plane [1, 2]. An early innovator in the control plane was Amazon Web Services, where a tenant can create a "Virtual Private Cloud" [1] with their IP addresses without interfering with other tenants. In the data plane, there has been little comparable progress.

To make comparable progress, we posit that the provider should present a simple performance abstraction of a dedicated switch connecting a tenant's endpoints [3], independent of the underlying physical topology. The endpoints may be anywhere in the datacenter, but a tenant should be able to attain full line rate for any traffic pattern between its endpoints, constrained only by endpoint capacities. Bandwidth assurances to this tenant should suffer no negative impact from the behavior and churn of other tenants in the datacenter. This abstraction has been a consistent ask of enterprise customers considering moving to the cloud, as the enterprise mission demands a high degree of infrastructure predictability [4].

Is this abstraction realizable? EyeQ described in this paper attempts to deliver this abstraction for every tenant. This requires three key components of which EyeQ provides the final missing piece.

First, with little to no knowledge of tenant communication patterns, promising bandwidth guarantees to endpoints requires smart endpoint placement in a network with adequate capacity (for the worst case). Hence, topologies with bottlenecks between server–server ("east–west") traffic are undesirable. Fortunately, recent proposals [5, 6, 7] have demonstrated cost-effective means of building "high bisection bandwidth" network topologies. These topologies are realizable in practice (§2.3), and substantially lower the complexity of endpoint placement (§3.5) as server–server capacity is more uniform. Second, utilizing this high bisectional bandwidth requires effective traffic load balancing schemes to mitigate network hotspots. While today's routing protocols (e.g. Equal-Cost Multi-Path [8]) do a reasonable job of utilizing available capacity, there has

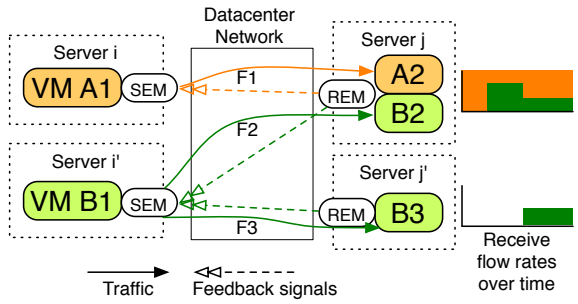


Figure 1: EyeQ's sender module (SEM) and receiver module (REM) work in a distributed fashion by exchanging feedback messages, to iteratively converge to bandwidth guarantees.

been continuous improvements on this front [9, 10, 11].

The third (and missing) piece, is a bandwidth arbitration mechanism that schedules tenant flows in accordance with the bandwidth guarantees, *even with misbehaving or malicious tenants.* Today, TCP's congestion control shares bandwidth equally across flows and is agnostic to tenant requirements, and thus falls short of predictably sharing bandwidth across tenants.

EyeQ, the main contribution of this paper, is a programmable bandwidth arbitration mechanism for the datacenter network. Our design is based on the key insight that by relieving the network's core of persistent congestion, we can partition bandwidth in a simple and distributed manner, completely at the edge. EyeQ uses server-to-server congestion control mechanisms to partition bandwidth locally at senders and receivers. The design is highly scalable and responsive and ensures bandwidth guarantees are met even in the presence of highly volatile traffic patterns. The congestion control mechanism pushes overloads back to the sources, while draining traffic at maximal rates. This ensures that network bandwidth is not wasted.

The EyeQ model. EyeQ allows administrators to configure a minimum and a maximum bandwidth to a VM's Virtual Network Interface Card (vNIC). The lower bound on bandwidth permits a work-conserving allocation among vNICs collocated on a physical machine.

The EyeQ arbitration mechanism. We explain the distributed mechanism using the example shown in Figure 1. VMs of tenants A and B are given a minimum bandwidth guarantee of 2Gb/s and 8Gb/s respectively. The first network flow F1 starts at VM A1 destined for A2. In the absence of contention, it is allocated the full line rate of 10Gb/s. While F1 is in progress, a second flow F2 starts at B1 destined for B2, creating congestion at server j. The Receiver EyeQ Module (REM) at j detects this contention for bandwidth, and uses end-to-end feedback to rate limit F1 to 2Gb/s, and F2 to 8Gb/s. Now, suppose flow F3 starts at VM B1 destined for B3. The Sender EyeQ Module (SEM) at server i' partitions its link bandwidth between F2 and F3 equally. Since this lowers the rate of F2 at server j to 5Gb/s, the REM at j will allocate the spare 3Gb/s bandwidth to F1 through subsequent feedback. In this way EyeQ recursively and distributedly schedules bandwidth across a network, to simultaneously maximize utilization, and meet bandwidth guarantees.

EyeQ is practical; the SEM and REM shim layers enforce traffic admission control *without* awareness of application traffic demands, traffic patterns, or transport protocol behavior (TCP/UDP) and *without* requiring any more support from network switches than what is already available today. We demonstrate this through extensive evaluations on real applications.

In summary, our main contributions are:

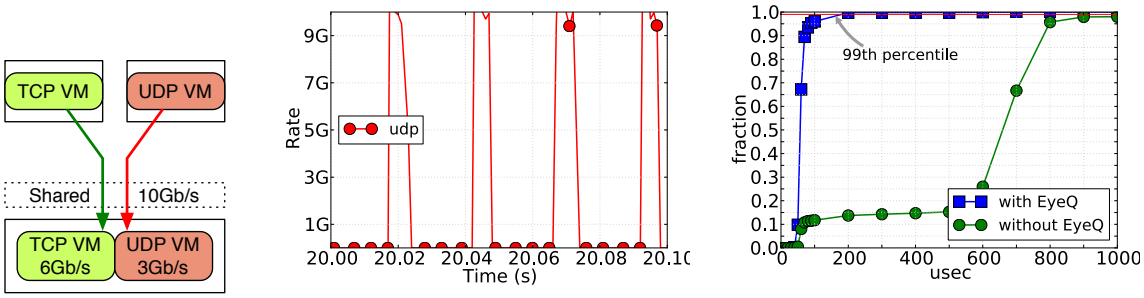
- The design of EyeQ that simultaneously achieves predictable and work-conserving bandwidth arbitration in a scalable fashion, completely from the network edge (host network stack, hypervisor, or NIC).
- An open implementation of EyeQ in software that scales to high line rates.
- An evaluation of EyeQ's feasibility at 10Gb/s on real applications.

The rest of the paper is organized as follows. We describe the nature of EyeQ's guarantees and discuss insights about network contention from a production cluster (§2) that motivate our design. We then delve into the design (§3), our software implementation (§4), and evaluation (§5) using micro- and macro-benchmarks. We summarize related work (§6) and conclude (§7).

We are committed to making our work easily available for reproducibility. Our implementation and evaluation scripts are online at `http://jvimal.github.com/eyeq`.

2 Predictable Bandwidth Partitioning

The goal of EyeQ is to schedule network traffic across a datacenter network such that it meets tenant endpoint bandwidth guarantees *over short intervals of time (e.g., a few milliseconds).* In this section, we define this notion of bandwidth guarantees more precisely and explain why bandwidth guarantees need to be met over short timescales. Then, we describe the key insight that makes EyeQ's simple design possible: The fact that the network's core in today's high bisection bandwidth datacenter networks can be kept free of persistent congestion. We show measurements from a Windows Azure production storage cluster that validate this claim.



(a) Topology setup.

(b) UDP burstiness at small timescales.

(c) Latency of TCP request response with and without EyeQ.

Figure 2: The UDP tenant bursts for 5ms and sleeps for 15ms, which even at 100ms timescales seems benign (2.5Gb/s or 25% utilization). These finer timescale interactions put a stringent performance requirement on the reaction times of any isolation mechanism, and mechanisms that react at large timescales may not see the big picture. EyeQ rate limits UDP over short timescales and improves the TCP tenant's median latency by over 10x. (There were no TCP timeouts in this experiment.)

2.1 EyeQ's bandwidth guarantees

EyeQ provides bandwidth guarantees for every endpoint (e.g., VM NIC) in order to mimic the performance of a dedicated switch for each tenant. The bandwidth guarantee for each endpoint is configured at provision time. The endpoints should be able to attain their guaranteed bandwidth as long as their traffic does not oversubscribe any endpoint's capacity.[1] For instance, if N VMs, each with 1Gb/s capacity, attempt to send traffic at full rate to a single 1Gb/s receiver, EyeQ only guarantees that the receiver (in aggregate) receives 1Gb/s of traffic. The excess traffic is dropped at the senders. Hence, EyeQ enforces traffic admissibility and only promises bandwidth guarantees for the *bottleneck port* (the receiver) and allocates bandwidth across senders in a max-min fashion.

There are different notions for bandwidth guarantees, ranging from exact rate and delay guarantees at the level of individual packets [13], to approximate "average rate" [14] guarantees over an acceptable interval of time. As observed in prior work [14], exact rate guarantees require per-endpoint queues and precise packet scheduling mechanisms [15, 16, 17] at every switch in the network. Such mechanisms are expensive to implement and are not available in switches today at a scale to isolate thousands of tenants and millions of VMs [18]. Hence, with EyeQ, we strive to attain average rate guarantees over an interval of time that is as short as possible.

2.2 Rate guarantees at short timescales

Datacenter traffic has been found to be highly volatile and bursty [5, 19, 20], leading to interactions at short timescales of a few milliseconds that adversely impact flow throughput and tail latency [21, 22, 23]. This is exacerbated by high network speeds and the use of shallow buffered commodity switches in datacenters. Today, a single large TCP flow is capable of causing congestion on its path in a matter of milliseconds, exhausting switch buffers [21, 24]. We refer the reader to [25] and our prior work [26] that demonstrate how bursty packet losses can adversely affect TCP's throughput.

The prior demonstrations highlight an artifact of TCP's behavior, but such interactions can also affect end-to-end *latency*, regardless of the transport protocol. To see this, consider a multi-tenant setting with a TCP and UDP tenant shown in Figure 2(a). Two VMs (one of each tenant) collocated on a physical machine receive traffic from their tenant VMs on other machines. Assume an administrator divides 9Gb/s of the access link bandwidth (at the receiver) between TCP and UDP tenants in the ratio 2:1 (the spare 1Gb/s or 10% bandwidth headroom is reserved to ensure good latency [22]). The UDP tenant transmits at an average rate of 2.5Gb/s by bursting in an ON-OFF fashion (ON at 10Gb/s for 5ms, OFF for 15ms). The TCP client issues back-to-back 1-byte requests over one connection and receives 1-byte responses from the server collocated with the UDP tenant.

Figure 2(c) shows the latency distribution of the TCP tenant with and without EyeQ. Though the average throughput of the UDP tenant (2.5Gb/s) is less than its allocated 3Gb/s, the TCP tenant's median and 99th percentile latency increases by over 10x. This is because of the congestion caused by the UDP tenant during the 5ms bursts at line rate, as shown in Figure 2(b). When EyeQ is enabled, it cuts off UDP's bursts at short timescales. We see that the latency with EyeQ is about 55μs, which

[1] This constraint is identical to what would occur with a dedicated switch, and is sometimes referred to as a *hose* constraint [12].

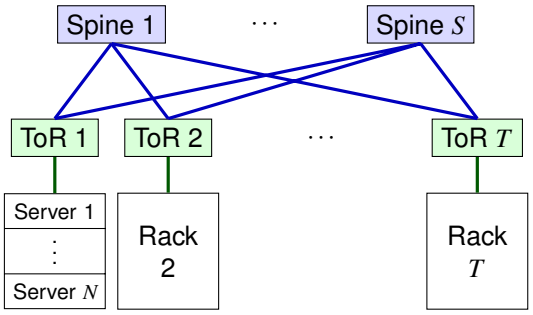


Figure 3: Emerging datacenter network architectures are over-subscribed only at the Top-of-Rack switches, which are connected to a spine layer that offers uniform bandwidth between racks. The over-subscription ratio is typically less than 3.

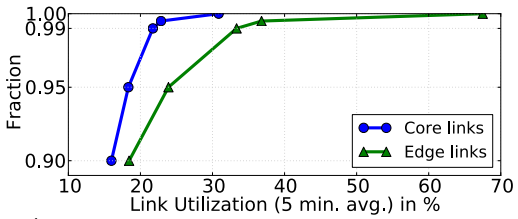


Figure 4: Utilization trends observed in a cluster running a multi-tenant storage service. The edge links exhibit higher peak and variance in link utilization compared to core links.

is close to bare-metal performance that we saw when running the TCP tenant without the UDP tenant.

Thus, any mechanism that only looks at average utilization over large timescales (e.g., over 100 milliseconds) fails to see the contention happening at finer timescales, which can substantially degrade both bandwidth and latency for contending applications. To address this challenge, EyeQ operates in the *dataplane* in a distributed fashion, and uses a responsive rate-based congestion control mechanism based on the Rate Control Protocol [27]. This enables EyeQ to quickly react to congestion in 100s of microseconds. We believe this timescale is short enough to at least protect tenants from persistent packet drops caused by other tenants as most datacenter switches have a few milliseconds worth of buffering.

2.3 The Fat and Flat Datacenter Network

In this section, we show how the high bisection bandwidth network architecture of a datacenter can simplify the task of bandwidth arbitration, which essentially boils down to managing network congestion wherever it occurs. In a flat datacenter network with little to no flow aggregation, congestion can occur everywhere, and therefore, switches need to be aware of thousands of tenants. Configuring every switch as tenants and their VMs come and go is unrealistic. We investigate where congestion actually occurs in a datacenter.

An emerging trend in datacenter network architecture is that the over-subscription ratio, typically less than 3:1, exists only at the Top-of-Rack (ToR) switches (Figure 3). Beyond the ToR switches, the network design eliminates any structural bottleneck, and offers uniform high capacity between racks in a cluster. To study where congestion occurs in such a topology, we collected link utilization statistics from Windows Azure's production storage cluster, whose network has a small oversubscription (less than 3:1). Figure 4 plots the top 10 percentiles of link utilization (averaged over 5 minute intervals) on edge and core links using data collected from 20 racks over the course of one week. The plot reveals two trends. First, we observe that edge links have higher peak link utilization. This suggests that persistent congestion manifests itself more often, and earlier, on server ports than the network core. Second, the variation of link utilization on core links is smaller. This suggests that the network core is evenly utilized and is free of persistent hot-spots.

The reason we observe this behavior is two fold. First, we observed that TCP is the dominant protocol in our datacenters [19, 21]. The nature of TCP's congestion control ensures that traffic is *admissible*, i.e., sources do not send more traffic (in aggregate) to a sink than the bottleneck capacity along the paths. In a high capacity fabric, the only bottlenecks are at over-subscription points—the server access links and the links between the ToRs and Spines—*provided* packets are optimally routed at other places. Second, datacenters today use Equal-Cost Multi-Path (ECMP) to randomize routing at the level of flows. In practice, ECMP is "good enough" to mitigate contention within the fabric, particularly when most flows are short-lived. While the above link utilizations reflect persistent congestion over 5 minute intervals, we conducted a detailed packet-level simulation study, and found that randomized per-packet routing can push even millisecond timescale congestion to the edge (§5.3).

Thus, if (a) the network has high bisection bandwidth, (b) the network employs randomized traffic routing, and (c) traffic is admissible, persistent congestion only occurs at the access links and not in the core. This observation guides our design in that it is sufficient if per-tenant state is pushed to the edge, where it is already available.

3 EyeQ Design

We now describe EyeQ's design in light of the observations in the §2. For ease of exposition, we first abstract the datacenter network as a single switch and describe

how EyeQ ensures rate guarantees to each endpoint connected to this switch. Then in §3.5, we explain how this design fits in a network of switches. Finally, we describe how endpoints that do not need guarantees can coexist.

If a single switch connects all servers, bandwidth contention happens only at the first-hop link connecting the sender to the switch, and the last-hop link connecting the switch to the receiver. Therefore, any contention is local to the servers, where the number of competing entities is small (typically 8–32 services/VMs per server). To resolve local contention between endpoints, two features are indispensable: (a) a mechanism that detects and accounts for contention, and (b) a mechanism that enforces rate limits on flows that violate their share. We now describe mechanisms to detect and resolve contention at senders and receivers.

3.1 Detecting and Resolving Contention

Senders. Contention between transmitters at the sender is straightforward to detect and resolve as the first point of contention is the server NIC. To resolve this and achive rate guarantees at the sender, EyeQ uses weighted fair queueing, where weights are set proportional to the endpoint's minimum bandwidth guarantees.

Receivers. However, contention at the receiver first happens *inside* the switch, and not at the receiving server. To see this, consider the example shown in Figure 2 where UDP generates highly bursty traffic that leads to 25% average utilization of the receiver link. When TCP begins to transmit packets, the link utilization soon approaches 100%, and packets are queued up in the limited buffer space inside the switch. If the switch does not differentially treat TCP and UDP packets, TCP's request/response experiences high latency.

Unfortunately, neither the sender nor receiver server has accurate, if any, visibility into this switch-internal contention, especially at timescales it takes to fill the switch packet buffers. These timescales can be very small as datacenter switches have limited packet buffers. Consider a scenario where two switch ports send data to a common port that has 1MB buffer. If each port starts sending at line rate, it takes just 800μs (at 10Gb/s) to fill the shared buffer inside the switch.

Fortunately, the scenario in Figure 2 offers an insight into the problem: contention happens when the link utilization, at short timescales, approaches its capacity. EyeQ therefore measures rate every 200μs and uses this information to rate limit flows before they cause further congestion. EyeQ stands to benefit if the network can further assist in quickly detecting any congestion, either using Explicit Congestion Notification (ECN) marks

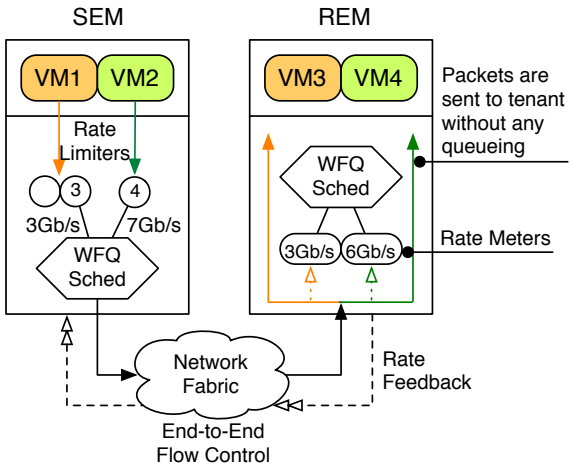


Figure 5: The design consists of a Sender EyeQ Module (SEM) and Receiver EyeQ Module (REM) at every end-host. The SEM consists of hierarchical rate limiters to enforce admission control, and a WRR scheduler to enforce minimum bandwidth guarantees. The REM consists of rate meters that detect and signal congestion. In tandem, the SEM and REM achieve end-to-end flow control.

from a shared queue when buffers exceed a configured queue occupancy, or using per-tenant dedicated queues only on the access link. But EyeQ does not need per-tenant network queues to function.

Resolving receiver contention. Unlike sender violation, performing both detection and rate limiting at the receiver is not effective, as rate limiting at the receiver can only control well-behaved TCP-like flows (by having them back off via drops). Unfortunately, VMs may use transport protocols such as UDP, which do not react to any downstream drops. EyeQ implements receiver-side detection, sender-side reaction. Specifically EyeQ detects bandwidth violation at the receiver using per-endpoint rate meters, and enforces rate limits at the senders using per-destination rate limiters. These per-destination rate limiters are programmed by *congestion feedback* generated by the receiver (§3.4).

In summary, EyeQ's design has two main components: (a) a rate meter at receivers that sends feedback to (b) rate limiters at senders. A combination of the above is needed to address both contention at the receiver indicated using feedback, as well as local contention at the sender. The rate limiters work in a distributed fashion using a control algorithm to iteratively converge to the 'right' rates.

3.2 Receiver EyeQ Module

The Receiver EyeQ Module (REM) consists of an RX scheduler and rate meters for every endpoint. The rate

meter is just as a byte counter that periodically (200μs) tracks the endpoint's receive rate, and invokes the RX scheduler which computes a capacity for each endpoint as $C_i = B_i \times C/(\sum_{j \in A} B_j)$. Here, A is the set of active VMs that are receiving traffic at non-zero rate, and B_i is the minimum bandwidth guarantee to VM i. Rate meters can be hierarchical; for example, a tenant can create rate meters to further split its capacity C_i across tenants communicating with it.

REM is clocked by incoming packets and measures each tenant's aggregate utilization every 200μs. A packet arrival triggers a feedback to source of the current packet. The feedback is a 16-bit value R computed by the rate meter using a control algorithm. To avoid generating excessive feedback, we send feedback only to the source address of packet sampled every 10kB of received data. This tends to choose senders that are communicating at high rates over a period of time.

This sampling also restricts the maximum bandwidth consumed by feedback. Since feedback packets are minimum sized packets (64 bytes), feedback traffic will not consume more than 64Mb/s (on a 10Gb/s link). This does not depend on the number of rate meters or senders transmitting to a single machine. We call this feedback packet a host-ACK, or HACK. The HACK is a special IP packet that is never communicated to the tenant; we picked the first unused IP protocol number (143) as a 'HACK.' The HACK encodes a 16-bit rate in its IPID field. However, this feedback can also be piggybacked on traffic to the source.

3.3 Sender EyeQ Module

To enforce traffic admission control, SEM uses multiple rate limiters organized in a hierarchical fashion. To see this hierarchy, consider the scenario in Figure 5. The root WRR scheduler schedules packet transmissions so that VM1 and VM2 get (say) equal share of the transmit bandwidth. To further ensure that traffic does not congest a destination, there are a set of rate limiters at the leaf level, one per *congested* destination. Per-destination rate limiters ensure that traffic to uncongested destinations are not head-of-line blocked by traffic to congested destinations. These per-destination rate limiters set their rate dictated by HACKs from receivers (§3.4). EyeQ associates traffic to a rate limiter only when the destination signals congestion through rate feedback. If a feedback is not received within 100 milliseconds, the rate limiter halves its rate until it hits 1Mb/s, to avoid congesting the network if the receiver is unresponsive (e.g. due to failures or network partitions).

3.4 Rate Control Loop

The heart of EyeQ's architecture is a rate control algorithm that computes the rates at which senders should converge to, to avoid overwhelming the receiver's capacity limits. The goal of this algorithm is to compute one rate R_i to which all flows of a tenant destined to endpoint i should be rate limited. If N senders all send long-lived flows, then R_i is simply C_i/N. In practice, N is hard to estimate as senders are in a constant state of flux, and not all of them may want to send traffic at rate R_i. Hence, we need a mechanism that can compute R_i without estimating the number of senders, or their demands. This makes the implementation practical, and more importantly, makes it possible to offload this functionality in hardware such as programmable NICs [28].

The control algorithm (operating at each endpoint) uses the measured receive rate y_i and the endpoint's allowed receive capacity C_i (determined by the RX scheduler) to compute a rate R_i that is advertised to senders communicating only with this endpoint. The basic idea is that the algorithm starts with an initial rate estimate R_i, and periodically corrects it based on observed y_i; if the incoming rate y_i is too small, it increases R_i, and if y_i is larger than C_i, it decreases R_i. This iterative procedure to compute R_i can be written as follows, taking care to keep R_i positive:

$$R_i \leftarrow R_i \left(1 - \alpha \cdot \frac{y_i - C_i}{C_i}\right)$$

The algorithm is a variant of the Rate Control Protocol (RCP) proposed in [27, 29], but there is an important difference. RCP's control algorithm operates on links in the network to split the link capacity among every flow in a max-min fashion. This achieves *per-flow* max-min fairness, and therefore, RCP suffers from the same problems as TCP. Instead, we operate the control algorithm in a hierarchical fashion. At the top level, the physical link capacity is divided by the RX scheduler into multiple virtual link capacities (C_i), one per VM, which *isolates* VMs from one another. Next, we operate the above algorithm independently on each virtual link.

The sensitivity of the algorithm is controlled by parameter α; higher values make the algorithm more aggressive in adjusting R_i. The above equation can be analyzed as follows. In the case where N flows traverse a single congested link of unit capacity, the rate evolution of R can be described in the standard form: [2] $z[n+1] = bz[n](1-z[n])$, where $z[n] = \left(\frac{b-1}{b}\right)\frac{R[n]}{R^*}$, $R^* = \frac{1}{N}$, and $b = 1+\alpha$. It can be shown that R^* is the only stable fixed point of the above recurrence if $1 < b < 3$, i.e.

[2] This is a standard non-linear one-dimensional dynamical system called the Logistic Map.

$0 < \alpha < 2$. By linearizing the recurrence about its fixed point, we can show that $R[n] \approx R^* + (R[0] - R^*)(1-\alpha)^n$. Therefore the system converges linearly. In practice, we found that high values of α lead to oscillations around R^* and therefore we recommend setting α conservatively to 0.5, for which $R[n]$ converges within 0.01% of R^* in about 30 iterations, irrespective of $R[0]$.

Though EyeQ makes an assumption about congestion free network core, ECN marks from network enable EyeQ to gracefully degrade in the presence of in-network congestion that can arise, for example, when links fail. In the rare event of persistent in-network congestion, we estimate the fraction of marked incoming packets as β and reduce R_i proportionally: $R_i \leftarrow R_i(1-\beta/2)$. This term β aids in reducing the rate of transmitting endpoints *only* in such transient cases. Though minimum bandwidth guarantees cannot be met in this case, it prevents starvation where one endpoint is completely dominated by another. In this case, bottleneck bandwidth is shared equally among all receiving endpoints.

We experimented with other control algorithms based on Data Center TCP (DCTCP) [21] and Quantized Congestion Notification (QCN) [30], and found that they have different convergence and stability properties. For example, DCTCP's convergence was on the order of 100–150ms, whereas QCN converged within 20–30ms. It is important that the control loop be fast and stable, to react to bursts without over-reaching, and RCP converges within a few milliseconds to the right rate. Since EyeQ computes rates every 200μs, the worst case convergence time (30 iterations) is 6ms. In practice, it is much faster.

3.5 EyeQ on a network

So far, we described EyeQ with the assumption that all end-hosts are connected to a single switch. A few steps must be taken to ensure EyeQ's design is directly applicable to networks that have a little over-subscription at the ToR switches (Figure 3). Clearly, if the policy is to guarantee minimum bandwidth to each VM, the cluster manager must ensure that capacity is not overbooked. This becomes simpler in such networks, where the core of the network is guaranteed to be congestion free, and hence admission control must only ensure that:

- The access links at end-hosts are not over-subscribed: i.e., the sum of bandwidth guarantees of VMs on a server is less than 10Gb/s.
- The ToR's uplink capacity is not over-subscribed: i.e., the sum of bandwidth guarantees of VMs under a ToR switch is less than the switch's total capacity to the Spine layer.

The above conditions ensure that VMs are guaranteed their bandwidth in the worst case when every VM needs it. The remaining capacity can be used for VMs that have no bandwidth requirements. Traffic from VMs that do not need bandwidth guarantees are mapped to a low priority, "best-effort" network class. This requires (i) a *one-time network configuration* of a low priority queue, which is easily possible in today's commodity switches, and (ii) end-hosts to mark packets so they can be classified to low priority network queues. This partitions the available bisection bandwidth across a class of VMs that need performance guarantees, and those that do not. As we saw in §3.4, EyeQ gracefully degrades in the presence of network congestion that can happen due to oversubscription, by sharing bandwidth equally among all receiving endpoints.

4 Implementation

EyeQ needs two components: rate limiters and rate metering. These components can be implemented in software, or hardware or a combination of two for optimum performance. In this paper, we present a full-software implementation of EyeQ's mechanisms, addressing the following challenges: (a) maintaining line rate performance at 10Gb/s while reacting quickly to deal with contentions at fine timescales, (b) co-existing with today's network stacks that use various offload techniques to speed up packet processing. EyeQ uses a combination of simple and well known techniques to reduce CPU overhead. We avoid writing to data structures shared across multiple CPUs to minimize cache misses. If sharing is inevitable, we minimize updates of shared data as much as possible through batching.

In untrusted environments, EyeQ is implemented in the trusted codebase at the (hypervisor or Dom0) virtual switch. In a non-virtualized, trusted environment, EyeQ resides in the network stack as a shim layer above the device driver. As a prototype, we implemented RX and TX processing for a VMSwitch filter driver for Windows Server 2008, and a kernel module for Linux which we use for all our experiments in this paper. The kernel module implements a queueing discipline (`qdisc`) in about 1900 lines of C code and about 700 lines of header files. We implemented a simple hash table based IP based classifier to identify endpoints. EyeQ hooks into the RX datapath using `netdev_rx_handler_register`.

4.1 Receiver EyeQ Module

The REM consists of rate meters, a scheduler and a HACK generator. A rate meter is created for each VM, and tracks the VM's receive rate in an integer. Clocked

by incoming packets, the scheduler determines each endpoint's allowed rate. The scheduler distributes the receive capacity among active endpoints, in accordance with their minimum bandwidth requirements.

At 10Gb/s, today's NICs use techniques such as Receive Side Scaling [31] to scale software packet processing by load balancing interrupts across CPU cores. A single, atomically updated byte counter in the critical path is a bottleneck and limits parallelism. To avoid such inefficiencies, we exploit the fact that today's NICs use interrupt coalescing to deliver multiple packets in a single interrupt, and therefore batch counter updates over 200μs time intervals. A smaller interval results in inaccurate rate measurement due to tiny bursts, and a larger interval decreases the rate meter's ability to detect short bursts that can cause interference. In a typical shallow buffered ToR switch that has a 1MB shared buffer, it takes 800μs to fill 1MB if two ports are sending at line rate to a common receiver. Thus, the choice of 200μs interval is to balance the ability to detect short bursts, and measure rate reasonably accurately.

4.2 Sender EyeQ Module

SEM consists of multiple TX-contexts, one per endpoint, that are isolated from one another. The SEM classifies packets to their corresponding TX-context. Each context has one root rate limiter, and a hash table of rate limiters keyed by IP destination d. The hash table stores the rate control state ($R_d^{(i)}$). Recall that rate enforcement is done hierarchically; leaf rate limiters enforce per-destination rates determined by end-to-end feedback loop, and the root rate limiter enforces a per-endpoint aggregate rate determined by the TX scheduler.

Rate limiters to IP destinations are created only on a need-to-rate limit basis. At start, packets to a destination are rate limited only at the root level. A per-destination rate limiter to a destination d is created, and added to the hierarchy, only on receiving a congestion feedback from the receiver d. Inactive rate limiters are garbage collected every few seconds. The TX WRR scheduler executes every 200μs and reassigns the total TX capacity to *active* endpoints, i.e., those that have a backlog of packets waiting to be transmitted in rate limiters.

Multi-queue rate limiter. The rate limiter is implemented as a token bucket, which has an associated linked-list (tail-drop) FIFO queue, a timer, a rate R and some tokens. This simple design can be inefficient, as a single queue rate limiter increases lock contention, which degrades performance significantly, as the queue is touched for every packet. Hence, we split the ideal rate limiter's FIFO queue into a per-CPU queue, and the total tokens into a local token count (t_c) on each CPU c. The value t_c is the number of bytes that Q_c can transmit without violating the global rate limit. Only if Q_c runs out of tokens to transmit the head of the queue, it grabs the rate limiter's lock to borrow all total tokens.

If the borrow fails due to lack of total tokens, the per-CPU queue is throttled and appended to a per-CPU list of backlogged queues. We found that having a timer for every rate limiter was very expensive. Therefore, a single per-CPU timer fires every 50μs and clocks only the *backlogged* rate limiters on that CPU. Decreasing the firing interval increases the precision of the rate limiter, but increases CPU overhead as it doubles the number of interrupts per second. In practice, we found that 50μs is sufficient. At 10Gb/s, at most 64kB can be transmitted every 50μs without violating rate constraints.

The rate limiter's per-CPU FIFO maximum queue size is restricted to 128kB, beyond which it back-pressures the network stack by refusing to accept more packets. While a TCP flow responds to this immediate feedback by stopping transmission, UDP applications may continue to send packets that will be dropped. Stopped TCP flows will be resumed by incoming ACKs.

Rate limiter accuracy. Techniques such as large segmentation offload (LSO) make it challenging to enforce rates precisely. With default configuration, the TCP stack can transmit data in 64kB chunks, which takes 51.2μs to transmit at 10Gb/s. If a flow is rate limited to 1Gb/s, the rate limiter would transmit one 64kB chunk every 512μs. This burstiness affects the accuracy with which the rate meter measures rates. To limit burstiness, we restrict the maximum LSO packet size to 32kB, which enables reasonably accurate rate metering at 256μs intervals. For rates less than 1Gb/s, the rate limiter selectively disables segmentation offload by splitting large packets into smaller chunks of at most the MTU (1500 bytes). This improves rate precision without incurring much CPU overhead. Limiting the size of an LSO packet also improves latency for short flows by reducing head of line blocking at the NIC.

5 Evaluation

We evaluate EyeQ to understand the following aspects:

- **Responsiveness**: We stress EyeQ's convergence times against a large burst of UDP streams and find that EyeQ converges within 5ms to protect a collocated TCP tenant.
- **CPU overhead**: At 10Gb/s, we evaluate the main overhead of EyeQ due to its rate limiters. We find it outperforms the software rate limiters in Linux.

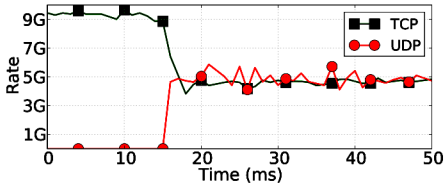


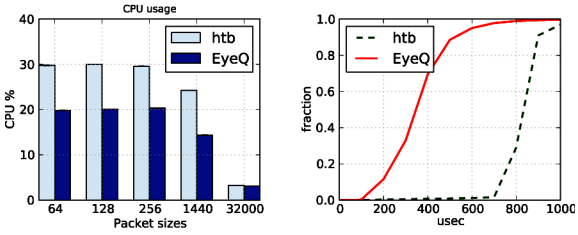
Figure 6: From rate samples taken every millisecond, we see that EyeQ converges within 5ms after UDP senders begin bursting at 14x *line rate*.

- **Flexibility**: Through EyeQ, we find that setting minimum bandwidth guarantees can directly affect the overall completion of a distributed all-to-all shuffle representative of Hadoop transfers.
- **Application performance**: In the presence of low volume, but highly bursty UDP traffic, we find that EyeQ is able to improve cluster utilization, and keep the 99.9th percentile response latency of a memcached cluster close to bare-metal performance.
- **Congestion in the fabric core**: Finally, we demonstrate using packet-level simulations that at high load, congestion even at millisecond timescales occurs more often at the edge than the network core.

Our evaluation cluster consists of 16 servers, each with a quad-core (2-way hyper-threaded) Intel Xeon 2.40Ghz processors. The servers run Linux 3.4 and are equipped with 10GbE NICs. All the servers are connected to a single shallow-buffered 24-port 10GbE switch, which has 20kB dedicated per-port buffer and 2MB shared memory. Since our switch lacks ECN support, we configure EyeQ to maintain a 10% bandwidth headroom at all times. In all our experiments, we enable Large Segmentation Offload, Receive Side Scaling (with 4 queues) and adaptive interrupt moderation. In our experiments, SEM (REM) use the source (destination) IP address to define endpoints in the transmit (receive) path, by creating subnets `11.0.T.0/24`, where T is the tenant ID.

5.1 Micro Benchmarks

Convergence times. We show a scenario that stresses the ability of EyeQ to quickly adapt and provide good isolation. There are two tenants: one TCP (1 sender and 1 receiver) and one UDP tenant: (N senders, and 1 receiver). Both receivers are collocated on the same host, but the senders are all collocated on different hosts. The TCP sender creates one long-lived TCP flow to the receiver. The UDP senders creates UDP streams, one each, simultaneously at t=30s, at maximum rate. As one would expect, without EyeQ, the TCP tenant starves while the UDP tenant runs active.



(a) CPU overhead at high load. (b) CDF of latency at low load.

Figure 7: EyeQ's rate limiter is multi-core aware, and is more efficient both in terms of CPU usage and packet latency, than today's Linux's Hierarchical Token Bucket `htb`.

When enabling EyeQ, the per-destination rate limiters are configured to start with an initial rate of 10Gb/s. This is representative of a possible worst case scenario for EyeQ, as it creates a sudden incast at rate 10N Gb/s that oversubscribes the access link by a factor of N. Each rate limiter starts at 10Gb/s and eventually has to converge to 5/N Gb/s. This initial burst eats into the headroom, and stresses the ability of EyeQ's convergence times. Figure 6 demonstrates the ability of EyeQ to enforce predictable performance, and provide a minimum bandwidth guarantee to TCP for N=14. EyeQ quickly converges within 5ms. Moreover, when the UDP tenant is not active, TCP is able to grab all 10Gb/s of bandwidth, which demonstrates the work-conserving nature of EyeQ, upto 90% of the capacity.

CPU overhead. Most of EyeQ's overhead is in rate limiting. We measure the overhead of a *single* EyeQ's rate limiter and contrast it to Linux's Hierarchical Token Bucket `htb`. Only for this test, we use a dual socket, 24 core Intel processor. To measure CPU overhead, we create a single rate limiter (at 5Gb/s), and generate traffic with varying packet sizes using 512 netperf processes. Figure 7(a) compares the CPU usage of `htb` with EyeQ, for varying packet sizes. We see that EyeQ's rate limiters have 1.5x lower CPU usage compared to `htb`. As one would expect, for a given rate, larger packet sizes implies smaller number of packets/second in software, and therefore the overhead is comparable for 32kB packets.

To measure latency overhead of `htb` due to locking, we ran 512 netperf processes, each generating back-to-back 1-byte requests and waiting for 1-byte responses. On a bare-metal system, this test generated about 10Mb/s of traffic, and thus the rate limiter configured at 5Gb/s should not have any effect. EyeQ's rate limiters work well, but with `htb`, we observe a 2.2x increase in the *median* per-transaction latency, as shown in Figure 7(b). Linux's `htb` acquires a single spinlock for each packet, and this increases the latency of enqueueing a packet into the rate limiter when the lock is heavily contended.

Recall that EyeQ requires a number of rate limiters that varies depending on the number of flows. In practice, the number of *active* flows (flows that have outstanding data) is typically less than a few 100 on a machine [5]. Nevertheless, we evaluated EyeQ's rate limiters by creating 20000 long lived flows that are assigned to a number of rate limiters in a round robin fashion. As we increased the number of rate limiters connected to the root (which is limited to 5Gb/s) from 1 to 1000 to 10000, we found that the CPU usage stays the same. This is because the net work output (packets per second) is the same in all cases, except for the (small) overhead involved in book keeping rate limiters.

5.2 Macro Benchmarks

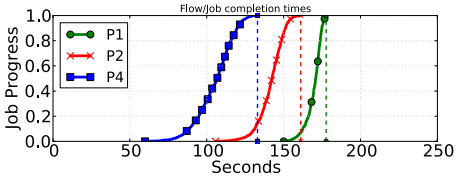
The micro-benchmarks show that EyeQ is efficient, and responsive in mitigating congestion. In this section, we explore the benefits of EyeQ on traffic characteristics of two real world applications: a long data shuffle (e.g. Hadoop) and memcached.

5.2.1 All-to-all data shuffle

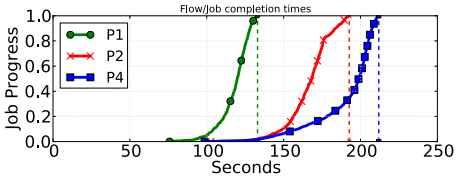
To mimic a Hadoop job's network traffic component, we generate an all to all traffic pattern of a sort job using a traffic generator.[3] Hadoop's reduce phase is bandwidth intensive and job completion times depend on the availability of bandwidth [32]. In the sort workload of S TB of data, using a cluster of N nodes involves roughly an equal amount of data shuffle between all pairs; in effect, $\frac{S}{N(N-1)}$ TB of data is shuffled between every pair of nodes. We use a TCP traffic generator to create long lived flows according to the above traffic pattern, and record the flow completion times of all the $N(N-1)$ flows. We then plot the CDF of flow completion times for every job to visualize its progress over time; the job is complete when the last flow completes. We make no optimizations to mitigate stragglers.

Multiple all-to-all shuffles. In this test, we run three collocated all-to-all shuffle jobs. Each job has 16 workers, one on each server in the cluster; and each server has three workers, one of each job. Each job has a varying degree of aggressiveness when consuming network bandwidth; job P_i ($i = 1,2,4$) creates i parallel TCP connections between each pair of its nodes, and each TCP connection an transfers equal amount of data. The jobs are all temporally and spatially collocated with each other and run a common 1 TB sort workload.

Figure 8(a) shows that jobs that open more TCP connections complete faster. However, EyeQ provides flexibility to explicitly configure job priorities, irrespective of the traffic, or protocol behavior. Figure 8(b) shows the job completion times if the lesser aggressive jobs are given higher priority; the priority can be inverted, and the job completion times reflect the change of priorities. The job priority is inverted by assigning minimum bandwidth guarantees B_i to jobs P_i that is inversely proportional to their aggressiveness; i.e., $B_1 : B_2 : B_4 = 4 : 2 : 1$. The final completion time in EyeQ increases from 180s to 210s, due to two reasons. First, EyeQ's congestion detectors maintain a 10% bandwidth headroom inorder to work at the end hosts without network ECN support. Second, the REM (§3.2) does not share bandwidth in a fine-grained, per-packet fashion, but over a 200μs time window. This leads to a small loss of utilization, when (say) P_1 is allocated some bandwidth but does not use it.



(a) *Without EyeQ*, job P4 creates 4 parallel TCP connections between its workers and completes faster.



(b) *With different minimum bandwidth guarantees*, EyeQ can directly affect the job completion times.

Figure 8: EyeQ's ability to differentially allocate bandwidth to all-to-all shuffle jobs can affect their completion times, and can be done at runtime, without reconfiguring jobs.

5.2.2 Memcached

Our final macro-evaluation is a scenario where a memcached tenant is collocated alongside an adversarial UDP tenant. The memcached tenant is a cluster consists of 16 processes: 4 memcached instances and 12 clients. Each process is located on a different host. At the start of the experiment, each cache instance allocates 8GB of memory, each client starts one thread per cache instance, and each thread opens 10 permanent TCP connections to its designated cache instance.

Throughput test. We generate an external load of about 288k requests/sec load balanced equally across all clients; at each client, the mean load is 6000 requests/sec to each cache instance. The clients generate SET re-

[3] We used a traffic generator as our disks could not sustain enough write throughput to saturate a 10Gb/s network.

	Latency percentiles (μs)		
Scenario	**50th**	**99th**	**99.9th**
Bare	98	370	666
Bare+EyeQ	100	333	630
Bare+UDP	4127	0.89×10^6	1.1×10^6
Bare+UDP+EyeQ	102	437	750

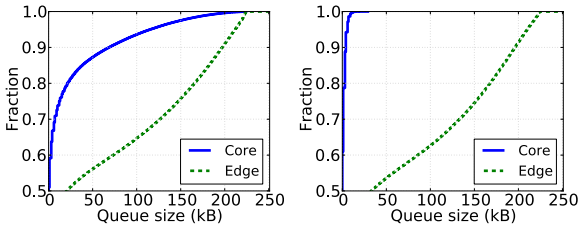
Table 1: Latency of memcached SET requests at low load (144k req/s). In all cases, the cluster throughput was the same, but EyeQ protects memcached from bursty traffic, bringing the 99.9th percentile latency closer to bare-metal performance.

quests of 6kB values and 32B keys and record the latency of each operation. We contrast the performance under four cases. First, we dedicate the cluster to the memcached tenant and establish baseline performance. The cluster was able to sustain the external load of 288k requests/sec. Second, we enable EyeQ on the same setup and found that EyeQ does not affect total throughput.

Third, we collocate memcached with a UDP tenant, by instantiating a UDP node on every end host. Each UDP node sends half-a-second burst of data to one other node (chosen in a round robin fashion), sleeping for half-a-second between bursts. Thus, the average utilization of UDP tenant is 5Gb/s. We chose this pattern as some cloud providers today allow a VM to burst at high rates for a few seconds before it is throttled. In this case, we find that the cluster was able to keep up only with 269k requests/sec which caused many responses *timed out* even though UDP tenant is consuming only 5Gb/s. Finally, we set equal bandwidth guarantees (5Gb/s) to both UDP and memcached tenant. We find that the cluster is can sustain the demand of 288k requests/sec. This shows that EyeQ is able to protect memcached tenant from the bursty UDP traffic.

Latency test. We over-provisioned the cluster by halving the external load (144k requests/sec). When memcached is collocated with UDP without EyeQ's protection, we observed that the cluster was able to meet its demand, but UDP was still able to affect the latency of memcached requests, increasing the 99th percentile latency by over three orders of magnitude. When enabled, EyeQ was able to protect the memcached tenant from fine-grained traffic bursts, and bring the 99.9th percentile latency to 750μs. The latency is still more than bare metal as the total load on the network is higher.

Takeaways. This experiment highlights a subtle point. Though we pay a 10% bandwidth price for low latency, EyeQ *improves* the net cluster utilization and tail latency performance. In a real setup, an unsuspecting client would pay money to spin up additional memcached instances to cope with the additional load. While this



(a) Per-flow ECMP. (b) Per-packet ECMP.

Figure 9: Queue occupancy distribution at the edge and core links, for two different routing algorithms. The maximum queue size is 225kB (150 packets). Large queue sizes indicate more congestion. In all cases, the network edge is more congested even at small timescales (1ms).

does increase revenue for providers, we believe they can earn more by using fewer resources to achieve the same level of performance. In datacenters, servers account for over 60% of the cost, but network accounts for only 10–15% [33]. With EyeQ, cloud operators can hope to achieve better CPU packing without worrying about network interference.

5.3 Congestion in the Fabric

In §2.3 we used link utilization from a production cluster as evidence that network congestion happens more often at the edge than the network core. These coarse-grained average link utilizations over five minute intervals show macroscopic trends, but it does not capture congestion at packet timescales. Using packet-level simulations in ns2 [34], we study the extent to which transient congestion can be mitigated at the network core using two routing algorithms: (a) per-flow ECMP and (b) a per-packet variant of ECMP. In the per-packet variant, each *packet*'s route is chosen uniformly at random among all next hops [35]. To cope with packet reordering, we increase TCP's duplicate ACK threshold.

We created a full bisection bandwidth topology of 144x10GbE hosts, 9 ToRs and 16 Spines as in our datacenters (Figure 3). Servers open TCP connections to every other server and generate traffic at an average rate of 10Gb/s$\times\lambda$, where λ is the offered load. Each server picks a random TCP connection to transmit data. Flow sizes are drawn from a distribution observed in a large datacenter [21]; the median, mean and 90th percentile flow sizes are 19kB, 2.4MB, 133kB respectively. We set queue sizes of all network queues to 150 packets and collect queue samples every 1ms. Figure 9 shows queue occupancy percentiles at the edge and core when $\lambda = 0.9$.

Even at high load, we observe that the core links are far less congested than the edge links. With per-packet ECMP, the *maximum* queue occupancy is less than 50kB

in the core links. All packet are dropped at the server access links. In all cases, we observed that per-packet ECMP practically eliminates in-network congestion *irrespective of traffic pattern*.

6 Related work

EyeQ's goals fall under the umbrella of Network Quality of Service (QoS), which has a rich history. Early QoS models [13, 36] and their implementations [15, 16, 36] focus on link-level and network-wide rate and delay guarantees for flows between a source and destination. Protocols such as Resource Reservation Protocol (RSVP) [37] reserve/relinquish resources across multiple links using such QoS primitives. Managing network state for every flow becomes untenable when there are a lot of flows. This led to approaches that relax strict guarantees for "average bandwidth," [14, 38, 39] while incurring lower state management overhead. A notable candidate is Core-Stateless Fair Queueing (CSFQ) that distributes state between the network core and the network edge, relying on flow aggregation at edge routers. In a flat datacenter network, tenant VMs are distributed across racks for availability, and hence there is little to no flow aggregation. OverQoS [40] provided an abstraction of a controlled loss virtual link with statistical bandwidth guarantees, between two nodes on an overlay network; this "pipe" model requires customers to specify bandwidth requirements between all communicating pairs, in contrast to a hose model [12].

Among recent approaches, Seawall [18] shares bottleneck capacity across competing source VMs (relative to their weights). This notion of sharing lacks predictability, as a tenant can grab more bandwidth by launching more source VMs. Oktopus [3] argues for predictability by enforcing a static hose model using rate limiters. It computes rates using a pseudo-centralized mechanism, where VMs communicate their pairwise bandwidth consumption to a tenant-specific centralized coordinator. This control plane overhead limits reaction times to about 2 seconds. However, as we have seen (§2.2), any isolation mechanism has to react quickly to be effective. SecondNet [41] is limited to providing static bandwidth reservations between pairs of VMs. In contrast to Oktopus and SecondNet, EyeQ supports both static and work conserving bandwidth allocations.

The closest related work to EyeQ is Gatekeeper [42], which also argues for predictable bandwidth allocation, and uses congestion control to provide rate guarantees to VMs. While the high level architecture is similar, Gatekeeper lacks details on the system design, especially the rate control mechanism, which is critical to providing bandwidth guarantees at short timescales. Gatekeeper's evaluation is limited to static scenarios with long lived flows. Moreover, Gatekeeper uses Linux's hierarchical token bucket, which incurs high overhead at 10Gb/s.

FairCloud [43] explored fundamental trade-offs between network utilization, min-guarantees and payment proportionality, for a number of sharing policies. FairCloud demonstrated the effect of such policies with per-flow queues in switches and CSFQ, which have limited or no support in today's commodity switches. However, the minimum-bandwidth guarantee that EyeQ supports conforms to FairCloud's 'Proportional-sharing on proximate links (PS-P)' sharing policy, which, as the authors demonstrate, outperform many other sharing policies. NetShare [44] used in-network weighted fair queueing to enforce bandwidth sharing among VMs. This approach, unfortunately, does not scale well due to the limited queues (8–64) per port.

The literature on congestion control mechanisms is vast; however, the fundamental unit of allocation is still per-flow, and therefore, the mechanisms are not adequate for network performance isolation. We refer the interested reader to [45] for a more comprehensive survey about recent efforts to address performance unpredictability in datacenter networks.

7 Concluding Remarks

In this paper, we presented EyeQ, a platform to enforce predictable network bandwidth sharing within the datacenter, using minimum bandwidth guarantees to endpoints. Our design and evaluation shows that a synthesis of well known techniques can lead to a simple and scalable design for network performance isolation. EyeQ is practical, and is deployable on today's, and next generation high speed datacenter networks with no changes to network hardware or applications. With EyeQ, providers can flexibly and efficiently apportion network bandwidth across tenants by giving each tenant endpoint a predictable minimum bandwidth guarantee, eliminating the problem of accidental, or malicious traffic interference.

Acknowledgments

We would like to thank the anonymous reviewers, Ali Ghodsi and our shepherd Lakshminarayanan Subramanian for their feedback and suggestions. The work at Stanford was funded by NSF FIA award CNS–1040190, by a gift from Google, and by DARPA CRASH award #N66001-10-2-4088. Opinions, findings, and conclusions do not necessarily reflect the views of the NSF or other sponsors.

References

[1] Amazon Virtual Private Cloud. `http://aws.amazon.com/vpc/`.

[2] Jayaram Mudigonda, Praveen Yalagandula, Jeff Mogul, Bryan Stiekes, and Yanick Pouffary. Netlord: a scalable multi-tenant network architecture for virtualized datacenters. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 62–73. ACM, 2011.

[3] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 242–253. ACM, 2011.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[5] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 51–62. ACM, 2009.

[6] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 39–50. ACM, 2009.

[7] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 75–86. ACM, 2008.

[8] C.E. Hopps. Analysis of an equal-cost multi-path algorithm. 2000.

[9] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 19–19, 2010.

[10] Theophilus Benson, Aditya Akella, Anees Shaikh, and Sambit Sahu. Cloudnaas: a cloud networking platform for enterprise applications. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 8. ACM, 2011.

[11] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 266–277. ACM, 2011.

[12] Nicholas G Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, Kadangode K Ramakrishnan, Jacobus E Van der Merwe, et al. A flexible model for resource management in virtual private networks. *ACM SIGCOMM Computer Communication Review*, 29(4):95–108, 1999.

[13] 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)*, 1(3):344–357, 1993.

[14] Rong Pan, Lee Breslau, Balaji Prabhakar, and Scott Shenker. Approximate fairness through differential dropping. In *ACM SIGCOMM Computer Communication Review*, volume 33, pages 23–39. ACM, 2003.

[15] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM Computer Communication Review*, volume 19, pages 1–12. ACM, 1989.

[16] Madhavapeddi Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *ACM SIGCOMM Computer Communication Review*, volume 25, pages 231–242. ACM, 1995.

[17] Jon CR Bennett and Hui Zhang. Wf2q: worst-case fair weighted fair queueing. In *INFOCOM'96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*, volume 1, pages 120–128. IEEE, 1996.

[18] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the data center network. In *USENIX NSDI*, volume 11, 2011.

[19] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 202–208. ACM, 2009.

[20] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review*, 40(1):92–99, 2010.

[21] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). *ACM SIGCOMM Computer Communication Review*, 40(4):63–74, 2010.

[22] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of USENIX NSDI conference*, 2012.

[23] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. Detail: Reducing the flow completion time tail in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 42(4):139–150, 2012.

[24] Mohammad Alizadeh, Berk Atikoglu, Abdul Kabbani, Ashvin Lakshmikantha, Rong Pan, Balaji Prabhakar, and Mick Seaman. Data center transport mechanisms: Congestion control theory and ieee standardization. In *Communication, Control, and Computing, 2008 46th Annual Allerton Conference on*, pages 1270–1277. IEEE, 2008.

[25] Aleksandar Kuzmanovic and Edward W Knightly. Low-rate tcp-targeted denial of service attacks: the shrew vs. the mice and elephants. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 75–86. ACM, 2003.

[26] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazieres, Balaji Prabhakar, Changhoon Kim, and Windows Azure. Eyeq: practical network performance isolation for the multi-tenant cloud. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 8–8. USENIX Association, 2012.

[27] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control. *ACM SIGCOMM Computer Communication Review*, 36(1):59–62, 2006.

[28] 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 *Proc. NSDI*, 2011.

[29] Frank Kelly, Gaurav Raina, and Thomas Voice. Stability and fairness of explicit congestion control with small buffers. *ACM SIGCOMM Computer Communication Review*, 38(3):51–62, 2008.

[30] Mohammad Alizadeh, Abdul Kabbani, Berk Atikoglu, and Balaji Prabhakar. Stability analysis of qcn: the averaging principle. *ACM SIGMETRICS Performance Evaluation Review*, 39(1):49–60, 2011.

[31] Guide to linux kernel network scaling. `http://code.google.com/p/kernel/wiki/NetScalingGuide`.

[32] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–16. USENIX Association, 2010.

[33] Internet-scale datacenter economics: Costs and opportunities. `http://mvdirona.com/jrh/TalksAndPapers/JamesHamilton_HPTS2011.pdf`.

[34] Steven McCanne, Sally Floyd, Kevin Fall, Kannan Varadhan, et al. Network simulator ns-2, 1997.

[35] Advait Dixit, Pawan Prakash, and Ramana Rao Kompella. On the efficacy of fine-grained traffic splitting protocols in data center networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 430–431. ACM, 2011.

[36] Ion Stoica, Hui Zhang, and TS Ng. *A hierarchical fair service curve algorithm for link-sharing, real-time and priority services*, volume 27. ACM, 1997.

[37] Lixia Zhang, Steve Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala. Rsvp: A new resource reservation protocol. *Network, IEEE*, 7(5):8–18, 1993.

[38] Ion Stoica, Scott Shenker, and Hui Zhang. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *ACM SIGCOMM Computer Communication Review*, volume 28, pages 118–130. ACM, 1998.

[39] Abdul Kabbani, Mohammad Alizadeh, Masato Yasuda, Rong Pan, and Balaji Prabhakar. Af-qcn: Approximate fairness with quantized congestion notification for multi-tenanted data centers. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pages 58–65. IEEE, 2010.

[40] Lakshminarayanan Subramanian, Ion Stoica, Hari Balakrishnan, and Randy Katz. Overqos: An overlay based architecture for enhancing internet qos. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, volume 1, pages 6–21, 2004.

[41] Chuanxiong Guo, Guohan Lu, Helen J Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference*, page 15. ACM, 2010.

[42] Henrique Rodrigues, Jose Renato Santos, Yoshio Turner, Paolo Soares, and Dorgival Guedes. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. *USENIX WIOV*, 2011.

[43] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. Faircloud: Sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 187–198. ACM, 2012.

[44] Sivasankar Radhakrishnan, Rong Pan, Amin Vahdat, George Varghese, et al. Netshare and stochastic netshare: predictable bandwidth allocation for data centers. *ACM SIGCOMM Computer Communication Review*, 42(3):5–11, 2012.

[45] Jeffrey C Mogul and Lucian Popa. What we talk about when we talk about cloud network performance. *ACM SIGCOMM Computer Communication Review*, 42(5):44–48, 2012.

Stronger Semantics for Low-Latency Geo-Replicated Storage

Wyatt Lloyd*, Michael J. Freedman*, Michael Kaminsky†, and David G. Andersen‡
*Princeton University, †Intel Labs, ‡Carnegie Mellon University

Abstract

We present the first scalable, geo-replicated storage system that guarantees low latency, offers a rich data model, and provides "stronger" semantics. Namely, all client requests are satisfied in the local datacenter in which they arise; the system efficiently supports useful data model abstractions such as column families and counter columns; and clients can access data in a causally-consistent fashion with read-only and write-only transactional support, even for keys spread across many servers.

The primary contributions of this work are enabling scalable causal consistency for the complex column-family data model, as well as novel, non-blocking algorithms for both read-only and write-only transactions. Our evaluation shows that our system, Eiger, achieves low latency (single-ms), has throughput competitive with eventually-consistent and non-transactional Cassandra (less than 7% overhead for one of Facebook's real-world workloads), and scales out to large clusters almost linearly (averaging 96% increases up to 128 server clusters).

1 Introduction

Large-scale data stores are a critical infrastructure component of many Internet services. In this paper, we address the problem of building a geo-replicated data store targeted at applications that demand fast response times. Such applications are now common: Amazon, EBay, and Google all claim that a slight increase in user-perceived latency translates into concrete revenue loss [25, 26, 41, 50].

Providing *low latency* to the end-user requires two properties from the underlying storage system. First, storage nodes must be near the user to avoid long-distance round trip times; thus, data must be replicated geographically to handle users from diverse locations. Second, the storage layer itself must be fast: client reads and writes must be local to that nearby datacenter and not traverse the wide area. Geo-replicated storage also provides the important benefits of availability and fault tolerance.

Beyond low latency, many services benefit from a *rich data model*. Key-value storage—perhaps the simplest data model provided by data stores—is used by a number of services today [4, 29]. The simplicity of this data model, however, makes building a number of interesting services overly arduous, particularly compared to the column-family data models offered by systems like BigTable [19] and Cassandra [37]. These rich data models provide hierarchical sorted column-families and numerical counters. Column-families are well-matched to services such as Facebook, while counter columns are particularly useful for numerical statistics, as used by collaborative filtering (Digg, Reddit), likes (Facebook), or re-tweets (Twitter).

Unfortunately, to our knowledge, no existing geo-replicated data store provides guaranteed low latency, a rich column-family data model, and *stronger consistency semantics*: consistency guarantees stronger than the weakest choice—eventual consistency—and support for atomic updates and transactions. This paper presents Eiger, a system that achieves all three properties.

The consistency model Eiger provides is tempered by impossibility results: the strongest forms of consistency—such as linearizability, sequential, and serializability—are impossible to achieve with low latency [8, 42] (that is, latency less than the network delay between datacenters). Yet, some forms of stronger-than-eventual consistency are still possible and useful, e.g., *causal consistency* [2], and they can benefit system developers and users. In addition, *read-only* and *write-only transactions* that execute a batch of read or write operations at the same logical time can strengthen the semantics provided to a programmer.

Many previous systems satisfy two of our three design goals. Traditional databases, as well as the more recent Walter [52], MDCC [35], Megastore [9], and some Cassandra configurations, provide stronger semantics and a rich data model, but cannot guarantee low latency. Redis [48], CouchDB [23], and other Cassandra configurations provide low latency and a rich data model, but not stronger semantics. Our prior work on COPS [43] supports low latency, some stronger semantics—causal consistency and read-only transactions—but not a richer data model or write-only transactions (see §7.8 and §8 for a detailed comparison).

A key challenge of this work is to meet these three goals while *scaling* to a large numbers of nodes in a

single datacenter, which acts as a single logical replica. Traditional solutions in this space [10, 12, 36], such as Bayou [44], assume a single node per replica and rely on techniques such as log exchange to provide consistency. Log exchange, however, requires serialization through a single node, which does not scale to multi-node replicas.

This paper presents Eiger, a scalable geo-replicated data store that achieves our three goals. Like COPS, Eiger tracks dependencies to ensure consistency; instead of COPS' dependencies on versions of keys, however, Eiger tracks dependencies on operations. Yet, its mechanisms do not simply harken back to the transaction logs common to databases. Unlike those logs, Eiger's operations may depend on those executed on other nodes, and an operation may correspond to a transaction that involves keys stored on different nodes.

Eiger's read-only and write-only transaction algorithms each represent an advance in the state-of-the-art. COPS introduced a read-only transaction algorithm that normally completes in one round of local reads, and two rounds in the worst case. Eiger's read-only transaction algorithm has the same properties, but achieves them using logical time instead of explicit dependencies. Not storing explicit dependencies not only improves Eiger's efficiency, it allows Eiger to tolerate long partitions between datacenters, while COPS may suffer a metadata explosion that can degrade availability.

Eiger's write-only transaction algorithm can atomically update multiple columns of multiple keys spread across multiple servers in a datacenter (i.e., they are atomic within a datacenter, but not globally). It was designed to coexist with Eiger's read-only transactions, so that both can guarantee low-latency by (1) remaining in the local datacenter, (2) taking a small and bounded number of local messages to complete, and (3) never blocking on any other operation. In addition, both transaction algorithms are general in that they can be applied to systems with stronger consistency, e.g., linearizability [33].

The contributions of this paper are as follows:

- The design of a low-latency, causally-consistent data store based on a column-family data model, including all the intricacies necessary to offer abstractions such as column families and counter columns.
- A novel non-blocking read-only transaction algorithm that is both performant and partition tolerant.
- A novel write-only transaction algorithm that atomically writes a set of keys, is lock-free (low latency), and does not block concurrent read transactions.
- An evaluation that shows Eiger has performance competitive to eventually-consistent Cassandra.

2 Background

This section reviews background information related to Eiger: web service architectures, the column-family data model, and causal consistency.

2.1 Web Service Architecture

Eiger targets large geo-replicated web services. These services run in multiple datacenters world-wide, where each datacenter stores a full replica of the data. For example, Facebook stores all user profiles, comments, friends lists, and likes at each of its datacenters [27]. Users connect to a nearby datacenter, and applications strive to handle requests entirely within that datacenter.

Inside the datacenter, client requests are served by a front-end web server. Front-ends serve requests by reading and writing data to and from storage tier nodes. Writes are asynchronously replicated to storage tiers in other datacenters to keep the replicas loosely up-to-date.

In order to scale, the storage cluster in each datacenter is typically partitioned across 10s to 1000s of machines. As a primitive example, Machine 1 might store and serve user profiles for people whose names start with 'A', Server 2 for 'B', and so on.

As a storage system, Eiger's *clients* are the front-end web servers that issue read and write operations on behalf of the human users. When we say, "a client writes a value," we mean that an application running on a web or application server writes into the storage system.

2.2 Column-Family Data Model

Eiger uses the column-family data model, which provides a rich structure that allows programmers to naturally express complex data and then efficiently query it. This data model was pioneered by Google's BigTable [19]. It is now available in the open-source Cassandra system [37], which is used by many large web services including EBay, Netflix, and Reddit.

Our implementation of Eiger is built upon Cassandra and so our description adheres to its specific data model where it and BigTable differ. Our description of the data model and API are simplified, when possible, for clarity.

Basic Data Model. The column-family data model is a "map of maps of maps" of named columns. The first-level map associates a key with a set of named column families. The second level of maps associates the column family with a set composed exclusively of either columns or super columns. If present, the third and final level of maps associates each super column with a set of columns. This model is illustrated in Figure 1: "Associations" are a column family, "Likes" are a super column, and "NSDI" is a column.

bool	⟵	batch_mutate	({key→mutation})
bool	⟵	atomic_mutate	({key→mutation})
{key→columns}	⟵	multiget_slice	({key, column_parent, slice_predicate})

Table 1: Core API functions in Eiger's column family data model. Eiger introduces `atomic_mutate` and converts `multiget_slice` into a read-only transaction. All calls also have an `actor_id`.

	User Data		Associations				
			Friends			Likes	
	ID	Town	Alice	Bob	Carol	NSDI	SOSP
Alice	1337	NYC	-	3/2/11	9/2/12	9/1/12	-
Bob	2664	LA	3/2/11	-	-	-	-
⋮							

Figure 1: An example use of the column-family data model for a social network setting.

Within a column family, each *location* is represented as a compound key and a single value, i.e., "Alice:Assocs:Friends:Bob" with value "3/2/11". These pairs are stored in a simple ordered key-value store. All data for a single row must reside on the same server.

Clients use the API shown in Table 1. Clients can insert, update, or delete columns for multiple keys with a `batch_mutate` or an `atomic_mutate` operation; each mutation is either an `insert` or a `delete`. If a column exists, an `insert` updates the value. Mutations in a `batch_mutate` appear independently, while mutations in an `atomic_mutate` appear as a single atomic group.

Similarly, clients can read many columns for multiple keys with the `multiget_slice` operation. The client provides a list of tuples, each involving a key, a column family name and optionally a super column name, and a slice predicate. The slice predicate can be a `(start,stop,count)` three-tuple, which matches the first `count` columns with names between `start` and `stop`. Names may be any comparable type, e.g., strings or integers. Alternatively, the predicate can also be a list of column names. In either case, a *slice* is a subset of the stored columns for a given key.

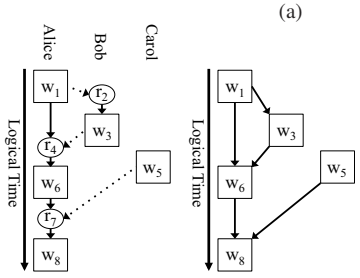
Given the example data model in Figure 1 for a social network, the following function calls show three typical API calls: updating Alice's hometown when she moves, ending Alice and Bob's friendship, and retrieving up to 10 of Alice's friends with names starting with B to Z.

```
batch_mutate(Alice→insert(UserData:Town=Rome))

atomic_mutate(Alice→delete(Assocs:Friends:Bob),
              Bob→delete(Assocs:Friends:Alice))

multiget_slice({Alice, Assocs:Friends, (B, Z, 10)})
```

Counter Columns. Standard columns are updated by `insert` operations that overwrite the old value. Counter columns, in contrast, can be commutatively updated using an `add` operation. They are useful for maintaining numerical statistics, e.g., a "liked_by_count" for NSDI (not shown in figure), without the need to carefully read-modify-write the object.

User	Op ID	Operation
Alice	w_1	insert(Alice, "-,Town", NYC)
Bob	r_2	get(Alice, "-,Town")
Bob	w_3	insert(Bob, "-,Town", LA)
Alice	r_4	get(Bob, "-,Town"))
Carol	w_5	insert(Carol, "Likes, NSDI", 8/31/12)
Alice	w_6	insert(Alice, "Likes, NSDI", 9/1/12)
Alice	r_7	get(Carol, "Likes, NSDI")
Alice	w_8	insert(Alice, "Friends, Carol", 9/2/12)

(a)



(b) (c)

Op	Dependencies
w_1	-
w_3	**w_1**
w_5	-
w_6	**w_3** w_1
w_8	**w_6** **w_5** w_3 w_1

(d)

Figure 2: (a) A set of example operations; (b) the graph of causality between them; (c) the corresponding dependency graph; and (d) a table listing nearest (bold), one-hop (underlined), and all dependencies.

2.3 Causal Consistency

A rich data model alone does not provide an intuitive and useful storage system. The storage system's consistency guarantees can restrict the possible ordering and timing of operations throughout the system, helping to simplify the possible behaviors that a programmer must reason about and the anomalies that clients may see.

The strongest forms of consistency (linearizability, serializability, and sequential consistency) are provably incompatible with our low-latency requirement [8, 42], and the weakest (eventual consistency) allows many possible orderings and anomalies. For example, under eventual consistency, after Alice updates her profile, she might not see that update after a refresh. Or, if Alice and Bob are commenting back-and-forth on a blog post, Carol might see a random non-contiguous subset of that conversation.

Fortunately, *causal consistency* can avoid many such inconvenient orderings, including the above examples, while guaranteeing low latency. Interestingly, the motivating example Google used in the presentation of their transactional, linearizable, and non-low-latency system Spanner [22]—where a dissident removes an untrustworthy person from his friends list and then posts politically sensitive speech—only requires causal consistency.

Causal consistency provides a partial order over operations in the system according to the notion of potential causality [2, 38], which is defined by three rules:

- **Thread-of-Execution.** An operation performed by a thread is causally after all of its previous ones.
- **Reads-From.** An operation that reads a value is causally after the operation that wrote the value.
- **Transitive-Closure.** If operation a is causally after b, and b is causally after c, then a is causally after c.

Figure 2 shows several example operations and illustrates their causal relationships. Arrows indicate the sink is causally after the source.

Write operations have *dependencies* on all other write operations that they are causally after. Eiger uses these dependencies to enforce causal consistency: It does not apply (commit) a write in a cluster until verifying that the operation's dependencies are *satisfied*, meaning those writes have already been applied in the cluster.

While the number of dependencies for a write grows with a client's lifetime, the system does not need to track every dependency. Rather, only a small subset of these, the *nearest dependencies*, are necessary for ensuring causal consistency. These dependencies, which have a longest path of one hop to the current operation, transitively capture all of the ordering constraints on this operation. In particular, because all non-nearest dependencies are depended upon by at least one of the nearest, if this current operation occurs after the nearest dependencies, then it will occur after all non-nearest as well (by transitivity). Eiger actually tracks *one-hop dependencies*, a slightly larger superset of nearest dependencies, which have a shortest path of one hop to the current operation. The motivation behind tracking one-hop dependencies is discussed in Section 3.2. Figure 2(d) illustrates the types of dependencies, e.g., w_6's dependency on w_1 is one-hop but not nearest.

3 Eiger System Design

The design of Eiger assumes an underlying partitioned, reliable, and linearizable data store inside of each datacenter. Specifically, we assume:

1. The keyspace is partitioned across logical servers.
2. Linearizability is provided inside a datacenter.
3. Keys are stored on logical servers, implemented with replicated state machines. We assume that a failure does not make a logical server unavailable, unless it makes the entire datacenter unavaible.

Each assumption represents an orthogonal direction of research to Eiger. By assuming these properties instead of specifying their exact design, we focus our explanation on the novel facets of Eiger.

Keyspace partitioning may be accomplished with consistent hashing [34] or directory-based approaches [6, 30]. Linearizability within a datacenter is achieved by partitioning the keyspace and then providing linearizability for each partition [33]. Reliable, linearizable servers can be implemented with Paxos [39] or primary-backup [3] approaches, e.g., chain replication [57]. Many existing systems [5, 13, 16, 54], in fact, provide all assumed properties when used inside a single datacenter.

3.1 Achieving Causal Consistency

Eiger provides causal consistency by explicitly checking that an operation's nearest dependencies have been applied before applying the operation. This approach is similar to the mechanism used by COPS [43], although COPS places dependencies on values, while Eiger uses dependencies on *operations*.

Tracking dependencies on operations significantly improves Eiger's efficiency. In the column family data model, it is not uncommon to simultaneously read or write many columns for a single key. With dependencies on values, a separate dependency must be used for each column's value and thus |column| dependency checks would be required; Eiger could check as few as one. In the worst case, when all columns were written by different operations, the number of required dependency checks degrades to one per value.

Dependencies in Eiger consist of a locator and a unique id. The *locator* is used to ensure that any other operation that depends on this operation knows which node to check with to determine if the operation has been committed. For mutations of individual keys, the locator is simply the key itself. Within a write transaction the locator can be any key in the set; all that matters is that each "sub-operation" within an atomic write be labeled with the same locator.

The *unique id* allows dependencies to precisely map to operations and is identical to the operation's timestamp. A node in Eiger checks dependencies by sending a `dep_check` operation to the node in its local datacenter that owns the locator. The node that owns the locator checks local data structures to see if has applied the operation identified by its unique id. If it has, it responds immediately. If not, it blocks the `dep_check` until it applies the operation. Thus, once all `dep_check`s return, a

server knows all causally previous operations have been applied and it can safely apply this operation.

3.2 Client Library

Clients access their local Eiger datacenter using a client library that: (1) mediates access to nodes in the local datacenter; (2) executes the read and write transaction algorithms; and, most importantly (3) tracks causality and attaches dependencies to write operations.[1]

The client library mediates access to the local datacenter by maintaining a view of its live servers and the partitioning of its keyspace. The library uses this information to send operations to the appropriate servers and sometimes to split operations that span multiple servers.

The client library tracks causality by observing a client's operations.[2] The API exposed by the client library matches that shown earlier in Table 1 with the addition of a `actor_id` field. As an optimization, dependencies are tracked on a per-user basis with the `actor_id` field to avoid unnecessarily adding thread-of-execution dependencies between operations done on behalf of different real-world users (e.g., operations issued on behalf of Alice are not entangled with operations issued on behalf of Bob).

When a client issues a write, the library attaches dependencies on its previous write and on all the writes that wrote a value this client has observed through reads since then. This *one-hop* set of dependencies is the set of operations that have a path of length one to the current operation in the causality graph. The one-hop dependencies are a superset of the nearest dependencies (which have a longest path of length one) and thus attaching and checking them suffices for providing causal consistency.

We elect to track one-hop dependencies because we can do so without storing any dependency information at the servers. Using one-hop dependencies slightly increases both the amount of memory needed at the client nodes and the data sent to servers on writes.[3]

3.3 Basic Operations

Eiger's basic operations closely resemble Cassandra, upon which it is built. The main differences involve the use of server-supplied logical timestamps instead of client-supplied real-time timestamps and, as described above, the use of dependencies and `dep_check`s.

Logical Time. Clients and servers in Eiger maintain a logical clock [38], and messages include a logical timestamp that updates these clocks. The clocks and timestamps provide a progressing logical time throughout the entire system. The low-order bits in each timestamps are set to the stamping server's unique identifier, so each is globally distinct. Servers use these logical timestamps to uniquely identify and order operations.

Local Write Operations. All three write operations in Eiger—`insert`, `add`, and `delete`—operate by replacing the current (potentially non-existent) column in a location. `insert` overwrites the current value with a new column, e.g., update Alice's home town from NYC to MIA. `add` merges the current counter column with the update, e.g., increment a liked-by count from 8 to 9. `delete` overwrites the current column with a tombstone, e.g., Carol is no longer friends with Alice. When each new column is written, it is *timestamped* with the current logical time at the server applying the write.

Cassandra atomically applies updates to a single row using snap trees [14], so all updates to a single key in a `batch_mutate` have the same timestamp. Updates to different rows on the same server in a `batch_mutate` will have different timestamps because they are applied at different logical times.

Read Operations. Read operations return the current column for each requested location. Normal columns return binary data. Deleted columns return an empty column with a deleted bit set. The client library strips deleted columns out of the returned results, but records dependencies on them as required for correctness. Counter columns return a 64-bit integer.

Replication. Servers replicate write operations to their *equivalent* servers in other datacenters. These are the servers that own the same portions of the keyspace as the local server. Because the keyspace partitioning may vary from datacenter to datacenter, the replicating server must sometimes split `batch_mutate` operations.

When a remote server receives a replicated `add` operation, it applies it normally, merging its update with the current value. When a server receives a replicated `insert` or `delete` operation, it compares the timestamps for each included column against the current column for each location. If the replicated column is logically newer, it uses the timestamp from the replicated column and otherwise overwrites the column as it would with a local write. That timestamp, assigned by the

[1] Our implementation of Eiger, like COPS before it, places the client library with the storage system client—typically a web server. Alternative implementations might store the dependencies on a unique node per client, or even push dependency tracking to a rich javascript application running in the client web browser itself, in order to successfully track web accesses through different servers. Such a design is compatible with Eiger, and we view it as worthwhile future work.

[2] Eiger can only track causality it sees, so the traditional criticisms of causality [20] still apply, e.g., we would not capture the causality associated with an out-of-band phone call.

[3] In contrast, our alternative design for tracking the (slightly smaller set of) nearest dependencies put the dependency storage burden on the servers, a trade-off we did not believe generally worthwhile.

datacenter that originally accepted the operation that wrote the value, uniquely identifies the operation. If the replicated column is older, it is discarded. This simple procedure ensures causal consistency: If one column is causally after the other, it will have a later timestamp and thus overwrite the other.

The overwrite procedure also implicitly handles *conflicting operations* that concurrently update a location. It applies the *last-writer-wins rule* [55] to deterministically allow the later of the updates to overwrite the other. This ensures that all datacenters converge to the same value for each column. Eiger could detect conflicts using previous pointers and then resolve them with application-specific functions similar to COPS, but we did not implement such conflict handling and omit details for brevity.

Counter Columns. The commutative nature of counter columns complicates tracking dependencies. In normal columns with overwrite semantics, each value was written by exactly one operation. In counter columns, each value was affected by many operations. Consider a counter with value 7 from +1, +2, and +4 operations. Each operation contributed to the final value, so a read of the counter incurs dependencies on all three. Eiger stores these dependencies with the counter and returns them to the client, so they can be attached to its next write.

Naively, every update of a counter column would increment the number of dependencies contained by that column *ad infinitum*. To bound the number of contained dependencies, Eiger structures the `add` operations occurring within a datacenter. Recall that all locally originating `add` operations within a datacenter are already ordered because the datacenter is linearizable. Eiger explicitly tracks this ordering in a new `add` by adding an *extra dependency* on the previously accepted `add` operation from the datacenter. This creates a single dependency chain that transitively covers all previous updates from the datacenter. As a result, each counter column contains at most one dependency per datacenter.

Eiger further reduces the number of dependencies contained in counter columns to the nearest dependencies *within* that counter column. When a server applies an `add`, it examines the operation's attached dependencies. It first identifies all dependencies that are on updates from other datacenters to this counter column. Then, if any of those dependencies match the currently stored dependency for another datacenter, Eiger drops the stored dependency. The new operation is causally after any local matches, and thus a dependency on it transitively covers those matches as well. For example, if Alice reads a counter with the value 7 and then increments it, her +1 is causally after all operations that commuted to create the 7. Thus, any reads of the resulting 8 would only bring a dependency on Alice's update.

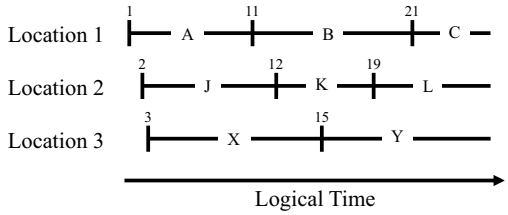


Figure 3: Validity periods for values written to different locations. Crossbars (and the specified numeric times) correspond to the earliest and latest valid time for values, which are represented by letters.

4 Read-Only Transactions

Read-only transactions—the only read operations in Eiger—enable clients to see a consistent view of multiple keys that may be spread across many servers in the local datacenter. Eiger's algorithm guarantees low latency because it takes at most two rounds of parallel non-blocking reads in the local datacenter, plus at most one additional round of local non-blocking checks during concurrent write transactions, detailed in §5.4. We make the same assumptions about reliability in the local datacenter as before, including "logical" servers that do not fail due to linearizable state machine replication.

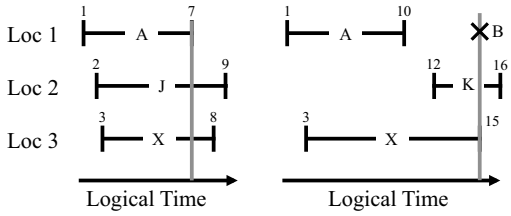
Why read-only transactions? Even though Eiger tracks dependencies to update each datacenter consistently, non-transactional reads can still return an inconsistent set of values. For example, consider a scenario where two items were *written* in a causal order, but read via two separate, parallel reads. The two reads could bridge the write operations (one occurring before either write, the other occurring after both), and thus return values that never actually occurred together, e.g., a "new" object and its "old" access control metadata.

4.1 Read-only Transaction Algorithm

The key insight in the algorithm is that *there exists a consistent result for every query at every logical time*. Figure 3 illustrates this: As operations are applied in a consistent causal order, every data location (key and column) has a consistent value at each logical time.

At a high level, our new read transaction algorithm marks each data location with validity metadata, and uses that metadata to determine if a first round of optimistic reads is consistent. If the first round results are not consistent, the algorithm issues a second round of reads that are guaranteed to return consistent results.

More specifically, each data location is marked with an *earliest valid time* (EVT). The EVT is set to the server's logical time when it locally applies an operation that writes a value. Thus, in an operation's accepting



(a) One Round Sufficient (b) Two Rounds Needed

Figure 4: Examples of read-only transactions. The effective time of each transaction is shown with a gray line; this is the time requested for location 1 in the second round in (b).

datacenter—the one at which the operation originated—the EVT is the same as its timestamp. In other datacenters, the EVT is later than its timestamp. In both cases, the EVT is the exact logical time when the value became visible in the local datacenter.

A server responds to a read with its currently visible value, the corresponding EVT, and its current logical time, which we call the *latest valid time* (LVT). Because this value is still visible, we know it is valid for at least the interval between the EVT and LVT. Once all first-round reads return, the client library compares their times to check for consistency. In particular, it knows all values were valid at the same logical time (i.e., correspond to a consistent snapshot) iff the maximum EVT $\leq$ the minimum LVT. If so, the client library returns these results; otherwise, it proceeds to a second round. Figure 4(a) shows a scenario that completes in one round.

The *effective time* of the transaction is the minimum LVT $\geq$ the maximum EVT. It corresponds both to a logical time in which all retrieved values are consistent, as well as the current logical time (as of its response) at a server. As such, it ensures freshness—necessary in causal consistency so that clients always see a progressing datacenter that reflects their own updates.

For brevity, we only sketch a proof that read transactions return the set of results that were visible in their local datacenter at the transaction's effective time, EffT. By construction, assume a value is visible at logical time t iff val.EVT $\leq t \leq$ val.LVT. For each returned value, if it is returned from the first round, then val.EVT $\leq$ maxEVT $\leq$ EffT by definition of maxEVT and EffT, and val.LVT $\geq$ EffT because it is not being requested in the second round. Thus, val.EVT $\leq$ EffT $\leq$ val.LVT, and by our assumption, the value was visible at EffT. If a result is from the second round, then it was obtained by a second-round read that explicitly returns the visible value at time EffT, described next.

4.2 Two-Round Read Protocol

A read transaction requires a second round if there does not exist a single logical time for which *all* values read in the first round are valid. This can only occur when there are concurrent updates being applied locally to the requested locations. The example in Figure 4(b) requires a second round because location 2 is updated to value K at time 12, which is not before time 10 when location 1's server returns value A.

```
function read_only_trans(requests):
  # Send first round requests in parallel
  for r in requests
    val[r] = multiget_slice(r)
  # Calculate the maximum EVT
  maxEVT = 0
  for r in requests
    maxEVT = max(maxEVT, val[r].EVT)
  # Calculate effective time
  EffT = ∞
  for r in requests
    if val[r].LVT ≥ maxEVT
      EffT = min(EffT, val[r].LVT)
  # Send second round requests in parallel
  for r in requests
    if val[r].LVT < EffT
       val[r] = multiget_slice_by_time(r, EffT)
  # Return only the requested data
  return extract_keys_to_columns(res)
```

Figure 5: Pseudocode for read-only transactions.

During the second round, the client library issues `multiget_slice_by_time` requests, specifying a read at the transaction's effective time. These reads are sent only to those locations for which it does not have a valid result, i.e., their LVT is earlier than the effective time. For example, in Figure 4(b) a `multiget_slice_by_time` request is sent for location 1 at time 15 and returns a new value B.

Servers respond to `multiget_slice_by_time` reads with the value that was valid at the requested logical time. Because that result may be different than the currently visible one, servers sometimes must store old values for each location. Fortunately, the extent of such additional storage can be limited significantly.

4.3 Limiting Old Value Storage

Eiger limits the need to store old values in two ways. First, read transactions have a timeout that specifies their maximum real-time duration. If this timeout fires—which happens only when server queues grow pathologically long due to prolonged overload—the client library restarts a fresh read transaction. Thus, servers only need to store old values that have been overwritten within this timeout's duration.

Second, Eiger retains only old values that could be requested in the second round. Thus, servers store only

values that are *newer* than those returned in a first round within the timeout duration. For this optimization, Eiger stores the last access time of each value.

4.4 Read Transactions for Linearizability

Linearizability (strong consistency) is attractive to programmers when low latency and availability are not strict requirements. Simply being linearizable, however, does not mean that a system is transactional: There may be no way to extract a mutually consistent set of values from the system, much as in our earlier example for read transactions. Linearizability is only defined on, and used with, operations that read or write a single location (originally, shared memory systems) [33].

Interestingly, our algorithm for read-only transactions works for fully linearizable systems, *without* modification. In Eiger, in fact, if all writes that are concurrent with a read-only transaction originated from the local datacenter, the read-only transaction provides a consistent view of that linearizable system (the local datacenter).

5 Write-Only Transactions

Eiger's write-only transactions allow a client to atomically write many columns spread across many keys in the local datacenter. These values also appear atomically in remote datacenters upon replication. As we will see, the algorithm guarantees low latency because it takes at most 2.5 message RTTs in the *local* datacenter to complete, no operations acquire locks, and all phases wait on only the previous round of messages before continuing.

Write-only transactions have many uses. When a user presses a save button, the system can ensure that all of her five profile updates appear simultaneously. Similarly, they help maintain symmetric relationships in social networks: When Alice accepts Bob's friendship request, both friend associations appear at the same time.

5.1 Write-Only Transaction Algorithm

To execute an `atomic_mutate` request—which has identical arguments to `batch_mutate`—the client library splits the operation into one sub-request per local server across which the transaction is spread. The library randomly chooses one key in the transaction as the *coordinator key*. It then transmits each sub-request to its corresponding server, annotated with the coordinator key.

Our write transaction is a variant of two-phase commit [51], which we call *two-phase commit with positive cohorts and indirection* (2PC-PCI). 2PC-PCI operates differently depending on whether it is executing in the original (or "accepting") datacenter, or being applied in the remote datacenter after replication.

There are three differences between traditional 2PC and 2PC-PCI, as shown in Figure 6. First, 2PC-PCI has only positive cohorts; the coordinator always commits the transaction once it receives a vote from all cohorts.[4] Second, 2PC-PCI has a different pre-vote phase that varies depending on the origin of the write transaction. In the accepting datacenter (we discuss the remote below), the client library sends each participant its sub-request directly, and this transmission serves as an implicit PREPARE message for each cohort. Third, 2PC-PCI cohorts that cannot answer a query—because they have voted but have not yet received the commit—ask the coordinator if the transaction is committed, effectively *indirecting* the request through the coordinator.

5.2 Local Write-Only Transactions

When a *participant* server, which is either the coordinator or a cohort, receives its transaction sub-request from the client, it prepares for the transaction by writing each included location with a special "pending" value (retaining old versions for second-round reads). It then sends a YESVOTE to the coordinator.

When the coordinator receives a YESVOTE, it updates its count of prepared keys. Once all keys are prepared, the coordinator commits the transaction. The coordinator's current logical time serves as the (global) timestamp and (local) EVT of the transaction and is included in the COMMIT message.

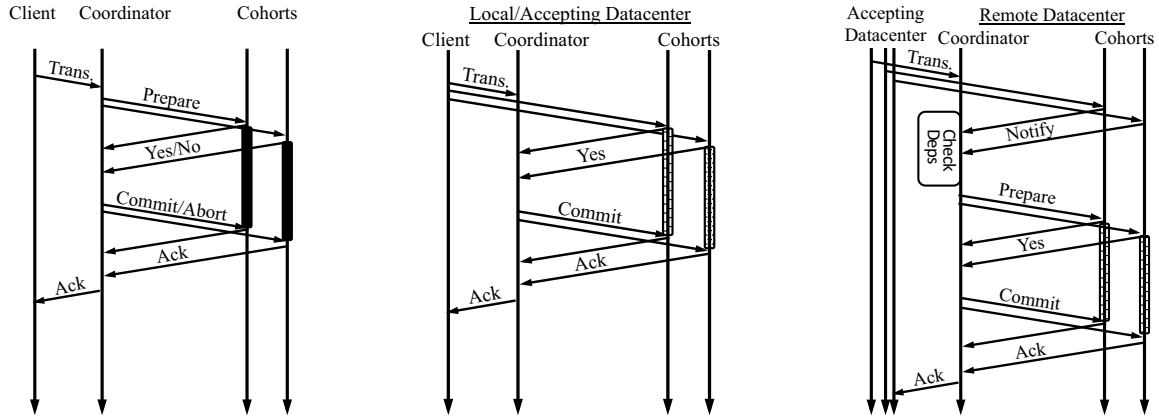
When a cohort receives a COMMIT, it replaces the "pending" columns with the update's real values, and ACKs the committed keys. Upon receiving all ACKs, the coordinator safely cleans up its transaction state.

5.3 Replicated Write-Only Transactions

Each transaction sub-request is replicated to its "equivalent" participant(s) in the remote datacenter, possibly splitting the sub-requests to match the remote key partitioning. When a cohort in a remote datacenter receives a sub-request, it sends a NOTIFY with the key count to the transaction coordinator in its datacenter. This coordinator issues any necessary `dep_check`s upon receiving its own sub-request (which contains the coordinator key). The coordinator's checks cover the entire transaction, so cohorts send no checks. Once the coordinator has received all NOTIFY messages and `dep_check`s responses, it sends each cohort a PREPARE, and then proceeds normally.

For reads received during the *indirection window* in which participants are uncertain about the status of a

[4] Eiger only has positive cohorts because it avoids all the normal reasons to abort (vote no): It does not have general transactions that can force each other to abort, it does not have users that can cancel operations, and it assumes that its logical servers do not fail.



(a) Traditional 2PC (b) Local Write-Only Txn (c) Replicated Write-Only Txn

Figure 6: Message flow diagrams for traditional 2PC and write-only transaction. Solid boxes denote when cohorts block reads. Striped boxes denote when cohorts will indirect a commitment check to the coordinator.

transaction, cohorts must query the coordinator for its state. To minimize the duration of this window, before preparing, the coordinator waits for (1) *all* participants to NOTIFY and (2) all `dep_check`s to return. This helps prevent a slow replica from causing needless indirection.

Finally, replicated write-only transactions differ in that participants do not always write pending columns. If a location's current value has a newer timestamp than that of the transaction, the validity interval for the transaction's value is empty. Thus, no read will ever return it, and it can be safely discarded. The participant continues in the transaction for simplicity, but does not need to indirect reads for this location.

5.4 Reads when Transactions are Pending

If a first-round read accesses a location that could be modified by a pending transaction, the server sends a special empty response that only includes a LVT (i.e., its current time). This alerts the client that it must choose an effective time for the transaction and send the server a second-round `multiget_slice_by_time` request.

When a server with pending transactions receives a `multiget_slice_by_time` request, it first traverses its old versions for each included column. If there exists a version valid at the requested time, the server returns it.

Otherwise, there are pending transactions whose *potential commit window* intersects the requested time and the server must resolve their ordering. It does so by sending a `commit_check` with this requested time to the transactions' coordinator(s). Each coordinator responds whether the transaction had been committed at that (past) time and, if so, its commit time.

Once a server has collected all `commit_check` responses, it updates the validity intervals of all versions of all relevant locations, up to at least the requested (effective) time. Then, it can respond to the `multiget_slice_by_time` message as normal.

The complementary nature of Eiger's transactional algorithms enables the atomicity of its writes. In particular, the single commit time for a write transaction (EVT) and the single effective time for a read transaction lead each to appear at a single logical time, while its two-phase commit ensures all-or-nothing semantics.

6 Failure

In this section, we examine how Eiger behaves under failures, including single server failure, meta-client redirection, and entire datacenter failure.

Single server failures are common and unavoidable in practice. Eiger guards against their failure with the construction of logical servers from multiple physical servers. For instance, a logical server implemented with a three-server Paxos group can withstand the failure of one of its constituent servers. Like any system built on underlying components, Eiger inherits the failure modes of its underlying building blocks. In particular, if a logical server assumes no more than f physical machines fail, Eiger must assume that within a single logical server no more than f physical machines fail.

Meta-clients that are the clients of Eiger's clients (i.e., web browsers that have connections to front-end web tier machines) will sometimes be directed to a different datacenter. For instance, a redirection may occur when there is a change in the DNS resolution policy of a service. When a redirection occurs during the middle of an active connection, we expect service providers to detect it using cookies and then redirect clients to their original datacenter (e.g., using HTTP redirects or triangle routing).

When a client is not actively using the service, however, policy changes that reassign it to a new datacenter can proceed without complication.

Datacenter failure can either be transient (e.g., network or power cables are cut) or permanent (e.g., datacenter is physically destroyed by an earthquake). Permanent failures will result in data loss for data that was accepted and acknowledged but not yet replicated to any other datacenter. The colocation of clients inside the datacenter, however, will reduce the amount of externally visible data loss. Only data that is not yet replicated to another datacenter, but has been acknowledged to both Eiger's clients and meta-clients (e.g., when the browser receives an Ajax response indicating a status update was posted) will be visibly lost. Transient datacenter failure will not result in data loss.

Both transient and permanent datacenter failures will cause meta-clients to reconnect to different datacenters. After some configured timeout, we expect service providers to stop trying to redirect those meta-clients to their original datacenters and to connect them to a new datacenter with an empty context. This could result in those meta-clients effectively moving backwards in time. It would also result in the loss of causal links between the data they observed in their original datacenter and their new writes issued to their new datacenter. We expect that transient datacenter failure will be rare (no ill effects), transient failure that lasts long enough for redirection to be abandoned even rarer (causality loss), and permanent failure even rarer still (data loss).

7 Evaluation

This evaluation explores the overhead of Eiger's stronger semantics compared to eventually-consistent Cassandra, analytically compares the performance of COPS and Eiger, and shows that Eiger scales to large clusters.

7.1 Implementation

Our Eiger prototype implements everything described in the paper as 5000 lines of Java added to and modifying the existing 75000 LoC in Cassandra 1.1 [17, 37]. All of Eiger's reads are transactional. We use Cassandra configured for wide-area eventual consistency as a baseline for comparison. In each local cluster, both Eiger and Cassandra use consistent hashing to map each key to a single server, and thus trivially provide linearizability.

In unmodified Cassandra, for a single logical request, the client sends all of its sub-requests to a single server. This server splits `batch_mutate` and `multiget_slice` operations from the client that span multiple servers, sends them to the appropriate server, and re-assembles the responses for the client. In Eiger, the client library handles this splitting, routing, and re-assembly directly, allowing Eiger to save a local RTT in latency and potentially many messages between servers. With this change, Eiger outperforms unmodified Cassandra in most settings. Therefore, to make our comparison to Cassandra fair, we implemented an analogous client library that handles the splitting, routing, and re-assembly for Cassandra. The results below use this optimization.

7.2 Eiger Overheads

We first examine the overhead of Eiger's causal consistency, read-only transactions, and write-only transactions. This section explains why each potential source of overhead does not significantly impair throughput, latency, or storage; the next sections confirm empirically.

Causal Consistency Overheads. Write operations carry *dependency metadata*. Its impact on throughput and latency is low because each dependency is 16B; the number of dependencies attached to a write is limited to its small set of one-hop dependencies; and writes are typically less frequent. Dependencies have no storage cost because they are not stored at the server.

Dependency check operations are issued in remote datacenters upon receiving a replicated write. Limiting these checks to the write's one-hop dependencies minimizes throughput degradation. They do not affect client-perceived latency, occuring only during asynchronous replication, nor do they add storage overhead.

Read-only Transaction Overheads. *Validity-interval metadata* is stored on servers and returned to clients with read operations. Its effect is similarly small: Only the 8B EVT is stored, and the 16B of metadata returned to the client is tiny compared to typical key/column/value sets.

If *second-round reads* were always needed, they would roughly double latency and halve throughput. Fortunately, they occur only when there are concurrent writes to the requested columns in the local datacenter, which is rare given the short duration of reads and writes.

Extra-version storage is needed at servers to handle second-round reads. It has no impact on throughput or latency, and its storage footprint is small because we aggressively limit the number of old versions (see §4.3).

Write-only Transaction Overheads. Write transactions *write columns twice*: once to mark them pending and once to write the true value. This accounts for about half of the moderate overhead of write transactions, evaluated in §7.5. When only some writes are transactional and when the writes are a minority of system operations (as found in prior studies [7, 28]), this overhead has a

	Latency (ms)			
	50%	90%	95%	99%
Reads				
Cassandra-Eventual	0.38	0.56	0.61	1.13
Eiger 1 Round	0.47	0.67	0.70	1.27
Eiger 2 Round	0.68	0.94	1.04	1.85
Eiger Indirected	0.78	1.11	1.18	2.28
Cassandra-Strong-A	85.21	85.72	85.96	86.77
Cassandra-Strong-B	21.89	22.28	22.39	22.92
Writes				
Cassandra-Eventual Cassandra-Strong-A	0.42	0.63	0.91	1.67
Eiger Normal	0.45	0.67	0.75	1.92
Eiger Normal (2)	0.51	0.79	1.38	4.05
Eiger Transaction (2)	0.73	2.28	2.94	4.39
Cassandra-Strong-B	21.65	21.85	21.93	22.29

Table 2: Latency micro-benchmarks.

small effect on overall throughput. The second write overwrites the first, consuming no space.

Many *2PC-PCI messages* are needed for the write-only algorithm. These messages add 1.5 local RTTs to latency, but have little effect on throughput: the messages are small and can be handled in parallel with other steps in different write transactions.

Indirected second-round reads add an extra local RTT to latency and reduce read throughput vs. normal second-round reads. They affect throughput minimally, however, because they occur rarely: only when the second-round read arrives when there is a not-yet-committed write-only transaction on an overlapping set of columns that prepared before the read-only transaction's effective time.

7.3 Experimental Setup

The first experiments use the shared VICCI testbed [45, 58], which provides users with Linux VServer instances. Each physical machine has 2x6 core Intel Xeon X5650 CPUs, 48GB RAM, and 2x1GigE network ports.

All experiments are between multiple VICCI sites. The latency micro-benchmark uses a minimal wide-area setup with a cluster of 2 machines at the Princeton, Stanford, and University of Washington (UW) VICCI sites. All other experiments use 8-machine clusters in Stanford and UW and an additional 8 machines in Stanford as clients. These clients fully load their local cluster, which replicates its data to the other cluster.

The inter-site latencies were 88ms between Princeton and Stanford, 84ms between Princeton and UW, and 20ms between Stanford and UW. Inter-site bandwidth was not a limiting factor.

Every datapoint in the evaluation represents the median of 5+ trials. Latency micro-benchmark trials are 30s, while all other trials are 60s. We elide the first and last quarter of each trial to avoid experimental artifacts.

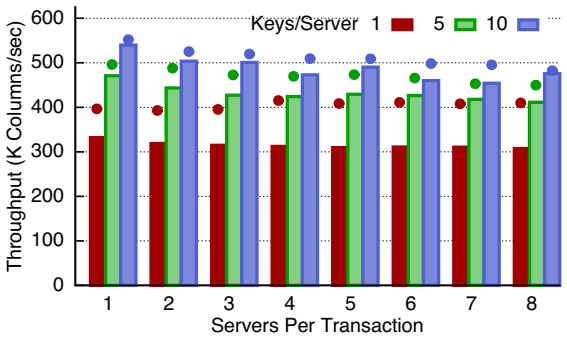


Figure 7: Throughput of an 8-server cluster for write transactions spread across 1 to 8 servers, with 1, 5, or 10 keys written per server. The dot above each bar shows the throughput of a similarly-structured eventually-consistent Cassandra write.

7.4 Latency Micro-benchmark

Eiger always satisfies client operations within a local datacenter and thus, fundamentally, is low-latency. To demonstrate this, verify our implementation, and compare with strongly-consistent systems, we ran an experiment to compare the latency of read and write operations in Eiger vs. three Cassandra configurations: eventual (R=1, W=1), strong-A (R=3, W=1), and strong-B (R=2, W=2), where R and W indicate the number of datacenters involved in reads and writes.[5]

The experiments were run from UW with a single client thread to isolate latency differences. Table 2 reports the median, 90%, 95%, and 99% latencies from operations on a single 1B column. For comparison, two 1B columns, stored on different servers, were also updated together as part of transactional and non-transactional "Eiger (2)" write operations.

All reads in Eiger—one-round, two-round, and worst-case two-round-and-indirected reads—have median latencies under 1ms and 99% latencies under 2.5ms. `atomic_mutate` operations are slightly slower than `batch_mutate` operations, but still have median latency under 1ms and 99% under 5ms. Cassandra's strongly consistent operations fared much worse. Configuration "A" achieved fast writes, but reads had to access all datacenters (including the ~84ms RTT between UW and Princeton); "B" suffered wide-area latency for both reads and writes (as the second datacenter needed for a quorum involved a ~20ms RTT between UW and Stanford).

7.5 Write Transaction Cost

Figure 7 shows the throughput of write-only transactions, and Cassandra's non-atomic batch mutates, when the

[5]Cassandra single-key writes are not atomic across different nodes, so its strong consistency requires read repair (write-back) and $R{>}N/2$.

Parameter	Range	Default	Facebook		
			50%	90%	99%
Value Size (B)	1-4K	128	16	32	4K
Cols/Key for Reads	1-32	5	1	2	128
Cols/Key for Writes	1-32	5	1	2	128
Keys/Read	1-32	5	1	16	128
Keys/Write	1-32	5		1	
Write Fraction	0-1.0	.1		.002	
Write Txn Fraction	0-1.0	.5		0 or 1.0	
Read Txn Fraction	1.0	1.0		1.0	

Table 3: Dynamic workload generator parameters. Range is the space covered in the experiments; Facebook describes the distribution for that workload.

keys they touch are spread across 1 to 8 servers. The experiment used the default parameter settings from Table 3 with 100% writes and 100% write transactions.

Eiger's throughput remains competitive with batch mutates as the transaction is spread across more servers. Additional servers only increase 2PC-PCI costs, which account for less than 10% of Eiger's overhead. About half of the overhead of write-only transactions comes from double-writing columns; most of the remainder is due to extra metadata. Both absolute and Cassandra-relative throughput increase with the number of keys written per server, as the coordination overhead remains independent of the number of columns.

7.6 Dynamic Workloads

We created a dynamic workload generator to explore the space of possible workloads. Table 3 shows the range and default value of the generator's parameters. The results from varying each parameter while the others remain at their defaults are shown in Figure 8.

Space constraints permit only a brief review of these results. Overhead decreases with increasing value size, because metadata represents a smaller portion of message size. Overhead is relatively constant with increases in the columns/read, columns/write, keys/read, and keys/write ratios because while the amount of metadata increases, it remains in proportion to message size. Higher fractions of write transactions (within an overall 10% write workload) do not increase overhead.

Eiger's throughput is overall competitive with the eventually-consistent Cassandra baseline. With the default parameters, its overhead is 15%. When they are varied, its overhead ranges from 0.5% to 25%.

7.7 Facebook Workload

For one realistic view of Eiger's overhead, we parameterized a synthetic workload based upon Facebook's production TAO system [53]. Parameters for value sizes,

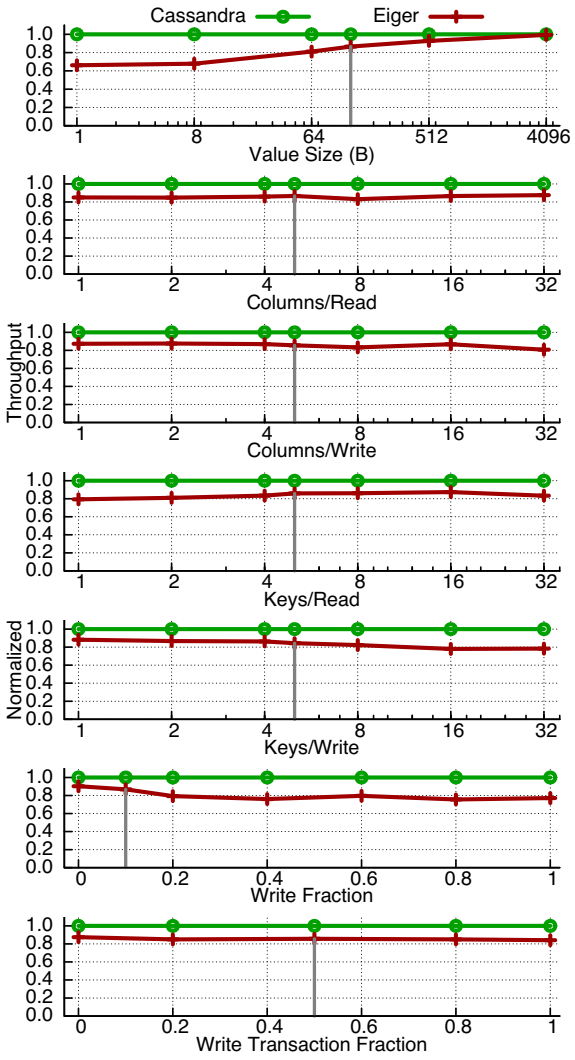


Figure 8: Results from exploring our dynamic-workload generator's parameter space. Each experiment varies one parameter while keeping all others at their default value (indicated by the vertical line). Eiger's throughput is normalized against eventually-consistent Cassandra.

columns/key, and keys/operation are chosen from discrete distributions measured by the TAO team. We show results with a 0% write transaction fraction (the actual workload, because TAO lacks transactions), and with 100% write transactions. Table 3 shows the heavy-tailed distributions' 50th, 90th, and 99th percentiles.

Table 4 shows that the throughput for Eiger is within 7% of eventually-consistent Cassandra. The results for 0% and 100% write transactions are effectively identical because writes are such a small part of the workload. For this real-world workload, Eiger's causal consistency and stronger semantics do not impose significant overhead.

	Ops/sec	Keys/sec	Columns/sec
Cassandra	23,657	94,502	498,239
Eiger	22,088	88,238	466,844
Eiger All Txns	22,891	91,439	480,904
Max Overhead	6.6%	6.6%	6.3%

Table 4: Throughput for the Facebook workload.

7.8 Performance vs. COPS

COPS and Eiger provide different data models and are implemented in different languages, so a direct empirical comparison is not meaningful. We can, however, intuit how Eiger's algorithms perform in the COPS setting.

Both COPS and Eiger achieve low latency around 1ms. Second-round reads would occur in COPS and Eiger equally often, because both are triggered by the same scenario: concurrent writes in the local datacenter to the same keys. Eiger experiences some additional latency when second-round reads are indirected, but this is rare (and the total latency remains low). Write-only transactions in Eiger would have higher latency than their non-atomic counterparts in COPS, but we have also shown their latency to be very low.

Beyond having write transactions, which COPS did not, the most significant difference between Eiger and COPS is the efficiency of read transactions. COPS's read transactions ("COPS-GT") add significant dependency-tracking overhead vs. the COPS baseline under certain conditions. In contrast, by tracking only one-hop dependencies, Eiger avoids the metadata explosion that COPS' read-only transactions can suffer. We expect that Eiger's read transactions would operate roughly as quickly as COPS' non-transactional reads, and the system as a whole would outperform COPS-GT despite offering both read- and write-only transactions and supporting a much more rich data model.

7.9 Scaling

To demonstrate the scalability of Eiger we ran the Facebook TAO workload on N client machines that are fully loading an N-server cluster that is replicating writes to another N-server cluster, i.e., the N=128 experiment involves 384 machines. This experiment was run on PRObE's Kodiak testbed [47], which provides an Emulab [59] with exclusive access to hundreds of machines. Each machine has 2 AMD Opteron 252 CPUS, 8GM RAM, and an InfiniBand high-speed interface. The bottleneck in this experiment is server CPU.

Figure 9 shows the throughput for Eiger as we scale N from 1 to 128 servers/cluster. The bars show throughput normalized against the throughput of the 1-server cluster. Eiger scales out as the number of servers increases, though this scaling is not linear from 1 to 8 servers/cluster.

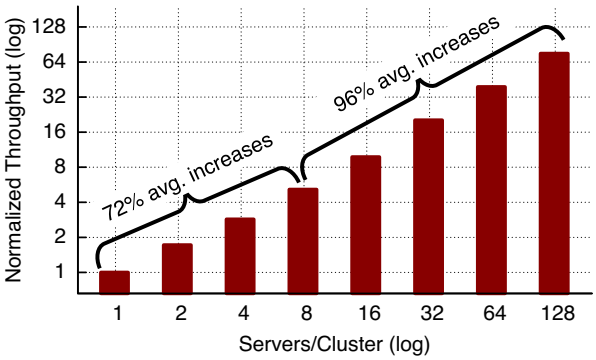


Figure 9: Normalized throughput of N-server clusters for the Facebook TAO workload. Bars are normalized against the 1-server cluster.

The 1-server cluster benefits from batching; all operations that involve multiple keys are executed on a single machine. Larger clusters distribute these multi-key operations over multiple servers and thus lose batching. This mainly affects scaling from 1 to 8 servers/cluster (72% average increase) and we see almost perfect linear scaling from 8 to 128 servers/cluster (96% average increase).

8 Related Work

A large body of research exists about stronger consistency in the wide area. This includes classical research about two-phase commit protocols [51] and distributed consensus (e.g., Paxos [39]). As noted earlier, protocols and systems that provide the strongest forms of consistency are provably incompatible with low latency [8, 42]. Recent examples includes Megastore [9], Spanner [22], and Scatter [31], which use Paxos in the wide-area; PNUTS [21], which provides sequential consistency on a per-key basis and must execute in a key's specified primary datacenter; and Gemini [40], which provides RedBlue consistency with low latency for its blue operations, but high latency for its globally-serialized red operations. In contrast, Eiger guarantees low latency.

Many previous system designs have recognized the utility of causal consistency, including Bayou [44], lazy replication [36], ISIS [12], causal memory [2], and PRACTI [10]. All of these systems require single-machine replicas (datacenters) and thus are not scalable.

Our previous work, COPS [43], bears the closest similarity to Eiger, as it also uses dependencies to provide causal consistency, and targets low-latency and scalable settings. As we show by comparing these systems in Table 5, however, Eiger represents a large step forward from COPS. In particular, Eiger supports a richer data model, has more powerful transaction support (whose algorithms also work with other consistency models), transmits and stores fewer dependencies, eliminates the need

	COPS	COPS-GT	Eiger
Data Model	Key Value	Key Value	**Column Fam**
Consistency	**Causal**	**Causal**	**Causal**
Read-Only Txn	No	**Yes**	**Yes**
Write-Only Txn	No	No	**Yes**
Txn Algos Use	-	Deps	**Logic. Time**
Deps On	Values	Values	**Operations**
Transmitted Deps	**One-Hop**	All-GarbageC	**One-Hop**
Checked Deps	One-Hop	**Nearest**	One-Hop
Stored Deps	**None**	All-GarbageC	**None**
GarbageC Deps	**Unneeded**	Yes	**Unneeded**
Versions Stored	**One**	Few	**Fewer**

Table 5: Comparing COPS and Eiger.

for garbage collection, stores fewer old versions, and is not susceptible to availability problems from metadata explosion when datacenters either fail, are partitioned, or suffer meaningful slow-down for long periods of time.

The database community has long supported consistency across multiple keys through general transactions. In many commercial database systems, a single primary executes transactions across keys, then lazily sends its transaction log to other replicas, potentially over the wide-area. In scale-out designs involving data partitioning (or "sharding"), these transactions are typically limited to keys residing on the same server. Eiger does not have this restriction. More fundamentally, the single primary approach inhibits low-latency, as write operations must be executed in the primary's datacenter.

Several recent systems reduce the inter-datacenter communication needed to provide general transactions. These include Calvin [56], Granola [24], MDCC [35], Orleans [15], and Walter [52]. In their pursuit of general transactions, however, these systems all choose consistency models that cannot guarantee low-latency operations. MDCC and Orleans acknowledge this with options to receive fast-but-potentially-incorrect responses.

The implementers of Sinfonia [1], TxCache [46], HBase [32], and Spanner [22], also recognized the importance of limited transactions. Sinfonia provides "mini" transactions to distributed shared memory and TXCache provides a consistent but potentially stale cache for a relational database, but both only considers operations within a single datacenter. HBase includes read- and write-only transactions within a single "region," which is a subset of the capacity of a single node. Spanner's read-only transactions are similar to the original distributed read-only transactions [18], in that they always take at least two rounds and block until all involved servers can guarantee they have applied all transactions that committed before the read-only transaction started. In comparison, Eiger is designed for geo-replicated storage, and its transactions can execute across large cluster of nodes, normally only take one round, and never block.

The widely used MVCC algorithm [11, 49] and Eiger maintain multiple versions of objects so they can provide clients with a consistent view of a system. MVCC provides full snapshot isolation, sometimes rejects writes, has state linear in the number of recent reads and writes, and has a sweeping process that removes old versions. Eiger, in contrast, provides only read-only transactions, never rejects writes, has at worst state linear in the number of recent writes, and avoids storing most old versions while using fast timeouts for cleaning the rest.

9 Conclusion

Impossibility results divide geo-replicated storage systems into those that can provide the strongest forms of consistency and those that can guarantee low latency. Eiger represents a new step forward on the low latency side of that divide by providing a richer data model and stronger semantics. Our experimental results demonstrate that the overhead of these properties compared to a non-transactional eventually-consistent baseline is low, and we expect that further engineering and innovations will reduce it almost entirely.

This leaves applications with two choices for geo-replicated storage. Strongly-consistent storage is required for applications with global invariants, e.g., banking, where accounts cannot drop below zero. And Eiger-like systems can serve all other applications, e.g., social networking (Facebook), encyclopedias (Wikipedia), and collaborative filtering (Reddit). These applications no longer need to settle for eventual consistency and can instead make sense of their data with causal consistency, read-only transactions, and write-only transactions.

Acknowledgments. The authors would like to thank the NSDI program committee and especially our shepherd, Ethan Katz-Bassett, for their helpful comments. Sid Sen, Ariel Rabkin, David Shue, and Xiaozhou Li provided useful comments on this work; Sanjeev Kumar, Harry Li, Kaushik Veeraraghavan, Jack Ferris, and Nathan Bronson helped us obtain the workload characteristics of Facebook's TAO system; Sapan Bhatia and Andy Bavier helped us run experiments on the VICCI testbed; and Gary Sandine and Andree Jacobson helped with the PRObE Kodiak testbed.

This work was supported by funding from National Science Foundation Awards CSR-0953197 (CAREER), CCF-0964474, MRI-1040123 (VICCI), CNS-1042537 and 1042543 (PRObE), and the Intel Science and Technology Center for Cloud Computing.

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM TOCS*, 27(3), 2009.

[2] M. Ahamad, G. Neiger, P. Kohli, J. Burns, and P. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9 (1), 1995.

[3] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. In *Conf. Software Engineering*, Oct. 1976.

[4] Amazon. Simple storage service. http://aws.amazon.com/s3/, 2012.

[5] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *SOSP*, Oct. 2009.

[6] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM TOCS*, 14(1), 1996.

[7] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[8] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM TOCS*, 12(2), 1994.

[9] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, Jan. 2011.

[10] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *NSDI*, May 2006.

[11] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computer Surveys*, 13(2), June 1981.

[12] K. P. Birman and R. V. Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Comp. Soc. Press, 1994.

[13] W. Bolosky, D. Bradshaw, R. Haagens, N. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *NSDI*, 2011.

[14] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *PPoPP*, Jan. 2010.

[15] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *SOCC*, 2011.

[16] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[17] Cassandra. http://cassandra.apache.org/, 2012.

[18] A. Chan and R. Gray. Implementing distributed read-only transactions. *IEEE Trans. Info. Theory*, 11(2), 1985.

[19] 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. *ACM TOCS*, 26(2), 2008.

[20] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *SOSP*, Dec. 1993.

[21] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*, Aug. 2008.

[22] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, Oct 2012.

[23] CouchDB. http://couchdb.apache.org/, 2012.

[24] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *USENIX ATC*, Jun 2012.

[25] P. Dixon. Shopzilla site redesign: We get what we measure. Velocity Conference Talk, 2009.

[26] eBay. Personal communication, 2012.

[27] Facebook. Personal communication, 2011.

[28] J. Ferris. The TAO graph database. CMU PDL Talk, April 2012.

[29] B. Fitzpatrick. Memcached: a distributed memory object caching system. http://memcached.org/, 2011.

[30] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, Oct. 2003.

[31] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in Scatter. In *SOSP*, Oct. 2011.

[32] HBase. http://hbase.apache.org/, 2012.

[33] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3), 1990.

[34] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC*, May 1997.

[35] T. Kraska, G. Pang, M. J. Franklin, and S. Madden. MDCC: Multi-data center consistency. *CoRR*, abs/1203.6049, 2012.

[36] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM TOCS*, 10(4), 1992.

[37] A. Lakshman and P. Malik. Cassandra – a decentralized structured storage system. In *LADIS*, Oct. 2009.

[38] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7), 1978.

[39] L. Lamport. The part-time parliament. *ACM TOCS*, 16(2), 1998.

[40] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, Oct 2012.

[41] G. Linden. Make data useful. Stanford CS345 Talk, 2006.

[42] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report TR-180-88, Princeton Univ., Dept. Comp. Sci., 1988.

[43] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, Oct. 2011.

[44] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, Oct. 1997.

[45] L. Peterson, A. Bavier, and S. Bhatia. VICCI: A programmable cloud-computing research testbed. Technical Report TR-912-11, Princeton Univ., Dept. Comp. Sci., 2011.

[46] D. R. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional consistency and automatic management in an application data cache. In *OSDI*, Oct. 2010.

[47] PRObE. http://www.nmc-probe.org/, 2013.

[48] Redis. http://redis.io/, 2012.

[49] D. P. Reed. *Naming and Synchronization in a Decentralized Computer Systems*. PhD thesis, Mass. Inst. of Tech., 1978.

[50] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. Velocity Conference Talk, 2009.

[51] D. Skeen. A formal model of crash recovery in a distributed system. *IEEE Trans. Info. Theory*, 9(3), May 1983.

[52] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, Oct. 2011.

[53] TAO. A read-optimized globally distributed store for social graph data. Under Submission, 2012.

[54] J. Terrace and M. J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *USENIX ATC*, June 2009.

[55] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Sys.*, 4(2), 1979.

[56] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, May 2012.

[57] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, Dec. 2004.

[58] VICCI. http://vicci.org/, 2012.

[59] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, Dec. 2002.

Bobtail: Avoiding Long Tails in the Cloud

Yunjing Xu, Zachary Musgrave, Brian Noble, Michael Bailey
University of Michigan
{yunjing,ztm,bnoble,mibailey}@umich.edu

Abstract

Highly modular data center applications such as Bing, Facebook, and Amazon's retail platform are known to be susceptible to long tails in response times. Services such as Amazon's EC2 have proven attractive platforms for building similar applications. Unfortunately, virtualization used in such platforms exacerbates the long tail problem by factors of two to four. Surprisingly, we find that poor response times in EC2 are a property of nodes rather than the network, and that this property of nodes is both pervasive throughout EC2 and persistent over time. The root cause of this problem is co-scheduling of CPU-bound and latency-sensitive tasks. We leverage these observations in Bobtail, a system that proactively detects and avoids these bad neighboring VMs without significantly penalizing node instantiation. With Bobtail, common communication patterns benefit from reductions of up to 40% in 99.9th percentile response times.

1 Introduction

Modern Web applications such as Bing, Facebook, and Amazon's retail platform are both interactive and dynamic. They rely on large-scale data centers with many nodes processing large data sets at less than human-scale response times. Constructing a single page view on such a site may require contacting hundreds of services [7], and a lag in response time from any one of them can result in significant end-to-end delays [1] and a poor opinion of the overall site [5]. Latency is increasingly viewed as *the* problem to solve [17, 18]. In these data center applications, the long tail of latency is of particular concern, with 99.9th percentile network round-trip times (RTTs) that are orders of magnitude worse than the median [1, 2, 29]. For these systems, one out of a thousand customer requests will suffer an unacceptable delay.

Prior studies have all targeted dedicated data centers. In these, network congestion is the cause of long-tail behavior. However, an increasing number of Internet-scale applications are deployed on commercial clouds such as Amazon's *Elastic Compute Cloud*, or EC2. There are a variety of reasons for doing so, and the recent EC2 outage [21] indicates that many popular online services rely heavily on Amazon's cloud. One distinction between dedicated data centers and services such as EC2 is the use of *virtualization* to provide for multi-tenancy with some degree of isolation. While virtualization does negatively impact latency overall [24, 19], little is known about the long-tail behavior on these platforms.

Our own large-scale measurements of EC2 suggest that median RTTs are comparable to those observed in dedicated centers, but the 99.9th percentile RTTs are up to four times longer. Surprisingly, we also find that nodes of the same configuration (and cost) can have long-tail behaviors that differ from one another by as much as an order of magnitude. This has important implications, as *good* nodes we measured can have long-tail behaviors better than those observed in dedicated data centers [1, 29] due to the difference in network congestion, while *bad* nodes are considerably worse. This classification appears to be a property of the nodes themselves, not data center organization or topology. In particular, bad nodes appear bad to all others, whether they are in the same or different data centers. Furthermore, we find that this property is relatively stable; good nodes are likely to remain good, and likewise for bad nodes within our five-week experimental period. Conventional wisdom dictates that larger (and therefore more expensive) nodes are not susceptible to this problem, but we find that larger nodes are not always better than smaller ones.

Using measurement results and controlled experiments, we find the root cause of the problem to be an interaction between virtualization, processor sharing, and non-complementary workload patterns. In particular, mixing latency-sensitive jobs on the same node with sev-

eral CPU-intensive jobs leads to longer-than-anticipated scheduling delays, despite efforts of the virtualization layer to avoid them. With this insight, we develop a simple yet effective test that runs locally on a newborn instance and screens between good and bad nodes. We measure common communication patterns [29] on live EC2 instances, and we show improvement in long-tail behavior of between 7% and 40%. While some limits to scale remain, our system effectively removes this first barrier.

2 Observations

Amazon's Elastic Compute Cloud, or EC2, provides dynamic, fine-grained access to computational and storage resources. Virtualization is a key enabler of EC2. We provide background on some of these techniques, and we describe a five-week measurement study of network latency in several different EC2 data centers. Such latency has both significant jitter and a longer tail than that observed in dedicated data centers. Surprisingly, the extra long tail phenomenon is a property of nodes, rather than topology or network traffic; it is pervasive throughout EC2 data centers and it is reasonably persistent.

2.1 Amazon EC2 Background

Amazon EC2 consists of multiple geographically separated *regions* around the world. Each region contains several *availability zones*, or AZs, that are physically isolated and have independent failure probabilities. Thus, one AZ is roughly equivalent to one data center. A version of the Xen hypervisor [3], with various (unknown) customizations, is used in EC2. A VM in EC2 is called an *instance*, and different types (e.g., small, medium, large, and extra large) of instances come with different performance characteristics and price tags. Instances within the same AZ or in different AZs within the same region are connected by a high-speed private network. However, instances within different regions are connected by the public Internet. In this paper, we focus on network tail latency between EC2 instances in the same region.

2.2 Measurement Methodology

Alizadeh *et al.* show that the internal infrastructure of Web applications is based primarily on TCP [1]. But instead of using raw TCP measurement, we use a TCP-based RPC framework called `Thrift`. `Thrift` is popular among Web companies like Facebook [20] and delivers a more realistic measure of network performance at the application level. To measure application-level round-trip-times (RTTs), we time the completion of synchronous RPC calls—`Thrift` adds about 60μs of overhead when compared to TCP SYN/ACK based raw RTT measurement. In addition, we use established TCP connections for all measurement, so the overhead of the TCP three-way handshake is not included in the RTTs.

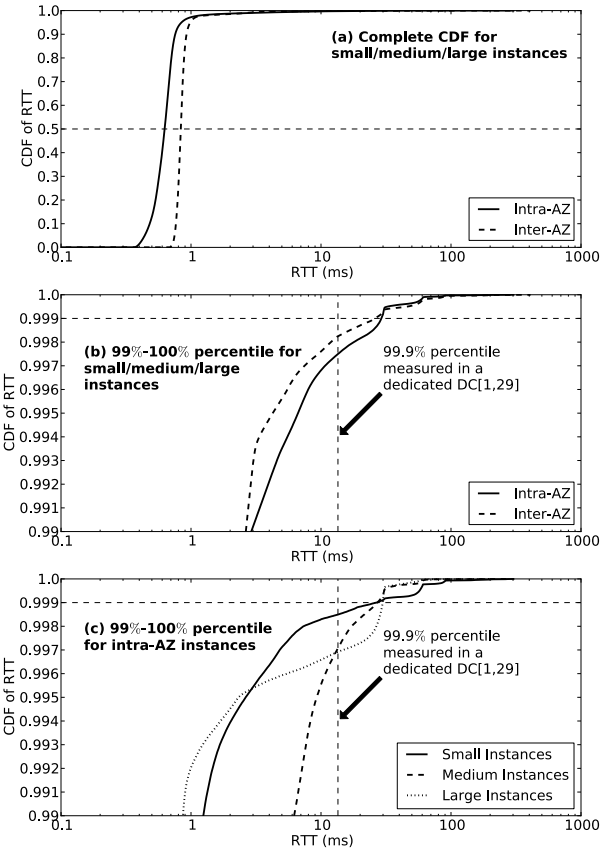


Figure 1: CDF of RTTs for various sized instances, within and across AZs in EC2, compared to measurements taken in a dedicated data center [1, 29]. While the median RTTs are comparable, the 99.9th percentiles in EC2 are twice as bad as in dedicated data centers. This relationship holds for all types of EC2 instances plotted.

2.3 Tail Latency Characterization

We focus on the tail of round-trip latency due to its disproportionate impact on user experience. Other studies have measured network performance in EC2, but they often use metrics like mean and variance to show jitter in network and application performance [24, 19]. While these measurements are useful for high-throughput applications like MapReduce [6], worst-case performance matters much more to applications like the Web that require excellent user experience [29]. Because of this, researchers use the RTTs at the 99th and 99.9th percentiles to measure flow tail completion times in dedicated data centers [1, 29].

We tested network latency in EC2's US east region for five weeks. Figure 1 shows CDFs for both a combination of small, medium, and large instances and for discrete sets of those instances. While (a) and (b) show aggregate measurements both within and across availability zones (AZs), (c) shows discrete measurements for three instance types within a specific AZ.

In Figure 1(a), we instantiated 20 instances of each type for each plot, either within a single AZ or across two AZs in the same region. We observe that median RTTs within a single AZ, at $\sim$0.6*ms*, compare well to those found within a dedicated data center at $\sim$0.4*ms* [1, 29], even though our measurement method adds 0.06*ms* of overhead. Inter-AZ measurements show a median RTT of under 1*ms*. However, distances between pairs of AZs may vary; measurements taken from another pair of AZs show a median RTT of around 2*ms*.

Figure 1(b) shows the 99th to 100th percentile range of (a) across all observations. Unfortunately, its results paint a different picture of latency measurements in Amazon's data centers. The 99.9th percentile of RTT measurements is *twice as bad* as the same metric in a dedicated data center [1, 29]. Individual nodes can have 99.9th percentile RTTs up to four times higher than those seen in such centers. Note that this observation holds for both curves; no matter whether the measurements are taken in the same data center or in different ones, the 99.9th percentiles are almost the same.

Medium, large, and extra large instances ostensibly offer better performance than their small counterparts. As one might expect, our measurements show that extra large instances do not exhibit the extra long tail problem ($<$ 0.9*ms* for the 99.9th percentile); but surprisingly, as shown in Figure 1(c), medium and large instances are susceptible to the problem. In other words, the extra long tail is not caused by a specific type of instance: all instance types shown in (c) are equally susceptible to the extra long tail at the 99.9th percentile. Note that all three lines in the figure intersect at the 99.9th line with a value of around 30ms. The explanation of this phenomenon becomes evident in the discussion of the root cause of the long tail problem in § 3.2.

To explore other factors that might create extra long tails, we launch 16 instances within the same AZ and measure the *pairwise* RTTs between each instance. Figure 2 shows measurement results at the 99.9th percentile in milliseconds. Rows represent source IP addresses, while columns represent destination IP addresses.

Were host location on the network affecting long tail performance, we would see a symmetric pattern emerge on the heat map, since network RTT is a symmetric measurement. Surprisingly, the heat map is asymmetric—there are vertical bands which do not correspond to reciprocal pairings. To a large degree, the destination host

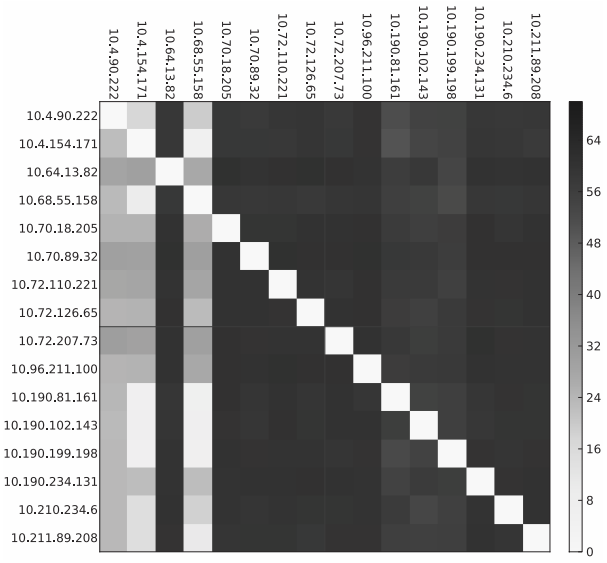


Figure 2: Heat map of the 99.9th percentile of RTTs, shown for 16 small pairwise instances in milliseconds. Bad instances, represented by dark vertical bands, are bad consistently. This suggests that the long tail problem is a property of specific nodes instead of the network.

controls whether a long tail exists. In other words, *the extra long tail problem in cloud environments is a property of nodes, rather than the network.*

Interestingly, the data shown in Figure 2 is not entirely bleak: there are *both* dark and light bands, so tail performance between nodes varies drastically. Commonly, RPC servers are allowed only 10*ms* to return their results [1]. Therefore, we refer to nodes that fulfill this service as *good* nodes, which appear in Figure 2 as light bands; otherwise, they are referred to as *bad* nodes. Under this definition, we find that RTTs at the 99.9th percentile can vary by *up to an order of magnitude* between good nodes and bad nodes. In particular, the bad nodes we measured can be two times worse than those seen in a dedicated DC [1, 29] for the 99.9th percentile. This is because the latter case's latency tail is caused by network congestion, whose worst case impact is bounded by the egress queue size of the bottleneck switch port, but the latency tail problem we study here is a property of nodes, and its worst case impact can be much larger than that caused by network queueing delay. This observation will become more clear when we discuss the root cause of the problem in § 3.2.

To determine whether bad nodes are a pervasive problem in EC2, we spun up 300 small instances in each of four AZs in the US east region. We measured all the nodes' RTTs (the details of the measurement benchmarks can be found in § 5.1) and we found 40% to 70% bad nodes within three of the four AZs.

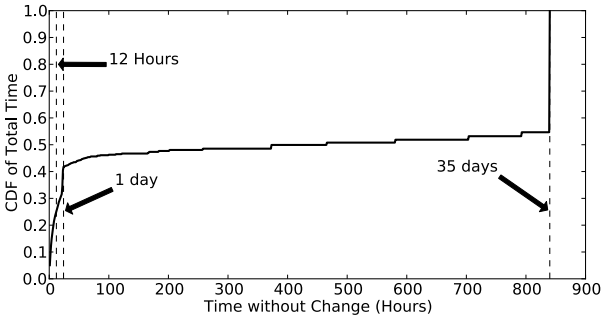


Figure 3: CDF for the time periods during which instances do not switch status between good and bad. This shows that properties of instances generally persist.

Interestingly, the remaining AZ sometimes does not return bad nodes; nevertheless, when it does, it returns 40% to 50% bad nodes. We notice that this AZ spans a smaller address space of only three /16 subnets compared to the others, which can span tens of /16 subnets. Also, its available CPU models are, on average, newer than those found in any of the other AZs; Ou *et al.* present similar findings [16], so we speculate that this data center is newly built and loaded more lightly than the others. We will discuss this issue further in conjunction with the root cause analysis in § 3.

We also want to explore whether the long latency tail we observe is a persistent problem, because it is a property defined by node conditions rather than transient network conditions. We conducted a five week experiment comprised of two sets of 32 small instances: one set was launched in equal parts from two AZs, and one set was launched from all four AZs. Within each set, we selected random pairs of instances and measured their RTTs throughout the five weeks. We observed how long instances' properties remain static—either good or bad without change—to show the persistence of our measurement results.

Figure 3 shows a CDF of these stable time periods; persistence follows if a large percentage of total instance time in the experiment is comprised of large time periods. We can observe that almost 50% of the total instance time no change has been witnessed, 60% of time involves at most one change per day, and 75% of time involves at most one change per 12 hours. This result shows that the properties of long tail network latency are generally persistent.

The above observation should be noted by the following: *every night*, *every instance* we observe in EC2 experiences an abnormally long latency tail for several minutes at midnight Pacific Time. For usually bad instances this does not matter; however, usually good instances are forced to change status at least once a day. Therefore, the figures we state above can be regarded as overestimating the frequency of changes. It also implies that the 50% instance time during which no change has been witnessed belongs to bad instances.

3 Root Cause Analysis

We know that the latency tail in EC2 is two to four times worse than that in a dedicated data center, and that as a property of nodes instead of the network it persists. Then, what is its root cause? Wang *et al.* reported that network latency in EC2 is highly variable, and they speculated that virtualization and processor sharing make up the root cause [24].

However, the coexistence of good and bad instances suggests that processor sharing under virtualization is *not sufficient* to cause the long tail problem by itself. We will show in this section that only *a certain mix of workloads* on shared processors can cause this problem, and we demonstrate the patterns of such a bad mix.

3.1 Xen Hypervisor Background

To fully understand the impact of processor sharing and virtual machine co-location on latency-sensitive workloads under Xen, we must first present some background on the hypervisor and its virtual machine scheduler. The Xen hypervisor [3] is an open source virtual machine monitor, and it is used to support the infrastructure of EC2 [24]. Xen consists of one privileged virtual machine (VM) called dom0 and multiple guest VMs called domUs. Its VM scheduler is credit-based [28], and by default it allocates 30*ms* of CPU time to each virtual CPU (VCPU); this allocation is decremented in 10*ms* intervals. Once a VCPU has exhausted its credit, it is not allowed to use CPU time unless no other VCPU has credit left; any VCPU with credit remaining has a higher priority than any without. In addition, as described by Dunlap [8], a lightly-loaded VCPU with excess credit may enter the `BOOST` state, which allows a VM to automatically receive first execution priority when it wakes due to an I/O interrupt event. VMs in the same `BOOST` state run in FIFO order. Even with this optimization, Xen's credit scheduler is known to be unfair to latency-sensitive workloads [24, 8]. As long as multiple VMs are sharing physical CPUs, one VM may need to wait tens of milliseconds to acquire a physical CPU if others are using it actively. Thus, when a VM handles RPC requests, certain responses will have to wait tens of milliseconds before being returned. This implies that *any* VM can exhibit a high maximum RTT.

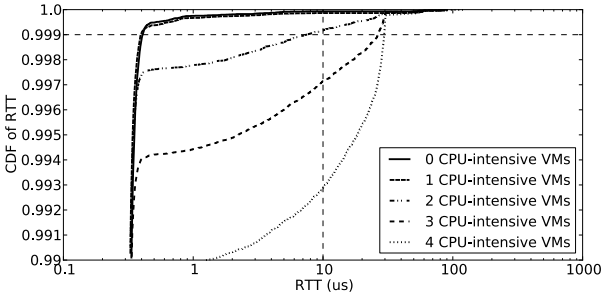


Figure 4: CDF of RTTs for a VM within controlled experiments, with an increasing number of co-located VMs running CPU-intensive workloads. Sharing does not cause extra long latency tails as long as physical cores outnumber CPU-intensive VMs, but once this condition no longer holds, the long tail emerges.

3.2 Root Cause Explained

If processor sharing under virtualization does not always cause the extra long tail problem, when does it? To answer this question, we conduct five controlled experiments with Xen.

On a four-core workstation running Xen 4.1, dom0 is pinned to two cores while guest VMs use the rest. In all experiments, five identically configured domUs share the remaining two physical cores; they possess equal weights of up to 40% CPU utilization each. Therefore, though domUs may be scheduled on either physical core, none of them can use more than 40% of a single core even if there are spare cycles. To the best of our knowledge, this configuration is the closest possible to what EC2 *small* instances use. Note that starting from Xen 4.2, a configurable rate limit mechanism is introduced to the credit scheduler [28]. In its default setting, a running VM cannot be preempted if it has run for less than 1ms. To obtain a result comparable to the one in this section using Xen 4.2 or newer, the rate limit needs to be set to its minimum of 0.1ms.

For this set of experiments, we vary the workload types running on five VMs sharing the local workstation. In the first experiment, we run the `Thrift` RPC server in all five guest VMs; we use another non-virtualized workstation in the same local network to make RPC calls to all five servers, once every two milliseconds, for 15 minutes. During the experiment, the local network is never congested. In the next four experiments, we replace the RPC servers on the guest VMs with a CPU-intensive workload, one at a time, until four guest VMs are CPU-intensive and the last one, called the *victim VM*, remains latency-sensitive.

Figure 4 shows the CDF of our five experiments' RTT distributions from the 99th to the 100th percentile for the victim VM. While four other VMs also run latency-sensitive jobs (zero VMs run CPU-intensive jobs), the latency tail up to the 99.9th percentile remains under 1*ms*. If one VM runs a CPU-intensive workload, this result does not change. Notably, even when the victim VM *does share* processors with one CPU-intensive VM and three latency-sensitive VMs, the extra long tail problem is *nonexistent*.

However, the 99.9th percentile becomes *five times* larger once two VMs run CPU-intensive jobs. This still qualifies as a good node under our definition ($< 10ms$), but the introduction of even slight network congestion could change that. To make matters worse, RTT distributions increase further as more VMs become CPU-intensive. Eventually, the latency-sensitive victim VM behaves just like the bad nodes we observe in EC2.

The results of the controlled experiments assert that virtualization and processor sharing are not sufficient to cause high latency effects across the entire tail of the RTT distribution; therefore, much of the blame rests upon co-located workloads. We show that having one CPU-intensive VM is acceptable; why does adding one more suddenly make things five times worse?

There are two physical cores available to guest VMs; if we have one CPU-intensive VM, the latency-sensitive VMs can be scheduled as soon as they need to be, while the single CPU-intensive VM occupies the other core. Once we reach two CPU-intensive VMs, it becomes possible that they occupy both physical cores concurrently while the victim VM has an RPC request pending. Unfortunately, the `BOOST` mechanism does not appear to let the victim VM preempt the CPU-intensive VMs often enough. Resulting from these unfortunate scenarios is an extra long latency distribution. In other words, *sharing does not cause extra long latency tails as long as physical cores outnumber CPU-intensive VMs; once this condition no longer holds, the long tail emerges.*

This set of controlled experiments demonstrates that a certain mix of latency-sensitive and CPU-intensive workloads on shared processors can cause the long tail problem, but a question remains: will all CPU-intensive workloads have the same impact? In fact, we notice that if the co-located CPU-intensive VMs in the controlled experiments always use 100% CPU time, the latency-sensitive VM *does not* suffer from the long tail problem—its RTT distribution is similar to the one without co-located CPU-intensive VMs; the workload we use in the preceding experiments actually uses about 85% CPU time. This phenomenon can be explained by the design of the `BOOST` mechanism. Recall that a VM waking up due to an interrupt may enter the `BOOST` state if it has credits remaining. Thus, if a VM doing mostly CPU-bound operations decides to accumulate scheduling credits, e.g., by using the `sleep` function call, it will also get `BOOST`ed after the `sleep` timer expires. Then, it may mo-

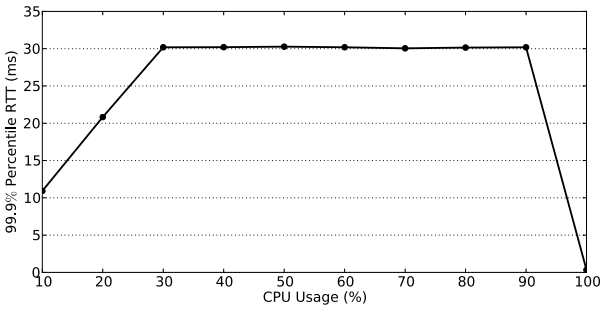


Figure 5: The relationship between the 99.9th percentile RTT for the latency-sensitive workload and the CPU usage of the CPU-bound workload in the neighboring VM.

nopolize the CPU until its credits are exhausted without being preempted by other BOOSTed VMs, some of which may be truly latency-sensitive.

In other words, the BOOST mechanism is only effective against the workloads that use almost 100% CPU time because such workloads exhaust their credits easily and BOOSTed VMs can then preempt them whenever they want. To study the impact of lower CPU usage, we conduct another controlled experiment by varying the CPU usage of the CPU-intensive workload from 10% to 100% and measuring the 99.9th percentile RTT of the latency-sensitive workload. We control the CPU usage using a command-line utility called `cpulimit` on a process that otherwise uses 100% CPU time; `cpulimit` pauses the target process periodically to adjust its average CPU usage. In addition, based on what we learned from the first set of controlled experiments, we only need to use one latency-sensitive VM to share a single CPU core with one CPU-intensive VM and allocate 50% CPU time to each one respectively.

Figure 5 shows the relationship between the 99.9th percentile RTT of the latency-sensitive workload and the CPU usage of the CPU-bound workload in the neighboring VM. Surprisingly, the latency tail is over 10ms even with 10% CPU usage, and starting from 30%, the tail latency is almost constant until 100%. This is because by default `cpulimit` uses a 10*ms* granularity: given X% CPU usage, it makes the CPU-intensive workload work Xms and pause $(100-X)ms$ in each 100*ms* window. Thus, when $X < 30$, the workload yields the CPU every Xms, so the 99.9th percentiles for the 10% and 20% cases are close to 10*ms* and 20*ms*, respectively; for $X \geq 30$, the workload keeps working for at least 30*ms*. Recall that the default time slice of the credit scheduler is 30*ms*, so the CPU-intensive workload cannot keep running for more than 30*ms* and we see the flat line in Figure 5. It also explains why the three curves in Figure 1(c) intersect at the 99.9th percentile line. The takeaway is that even if a workload uses as little as 10% CPU time on average, it still can cause a long latency tail to neighboring VMs by using large bursts of CPU cycles (e.g., 10*ms*). In other words, average CPU usage does not capture the intensity of a CPU-bound workload; *it is the length of the bursts of CPU-bound operations that matters*.

Now that we understand the root cause, we will examine an issue stated earlier: one availability zone in the US east region of EC2 has a higher probability of returning good instances than the other AZs. If we break down VMs returned from this AZ by CPU model, we find a higher likelihood of newer CPUs. These newer CPUs should be more efficient at context switching, which naturally shortens the latency tail, but what likely matters more is newer CPUs' possessing six cores instead of four, as in older CPUs that are more common in the other three data centers. One potential explanation for this is that the EC2 instance scheduler may not consider CPU model differences when scheduling instances sensitive to delays. Then, a physical machine with four cores is much more likely to be saturated with CPU-intensive workloads than a six-core machine. Hence, a data center with older CPUs is more susceptible to the problem. Despite this, our root cause analysis always applies, because we have observed that both good and bad instances occur regardless of CPU model; differences between them only change the likelihood that a particular machine will suffer from the long tail problem.

4 Avoiding the Long Tails

While sharing is inevitable in multi-tenant cloud computing, we set out to design a system, Bobtail, to find instances where processor sharing does not cause extra long tail distributions for network RTTs. Cloud customers can use Bobtail as a utility library to decide on which instances to run their latency-sensitive workloads.

4.1 Potential Benefits

To understand both how much improvement is possible and how hard it would be to obtain, we measured the impact of bad nodes for common communication patterns: sequential and partition-aggregation [29]. In the sequential model, an RPC client calls some number of servers in series to complete a single, timed observation. In the partition-aggregation model, an RPC client calls all workers in parallel for each timed observation.

For the sequential model, we simulate workflow completion time by sampling from the measured RTT distributions of good and bad nodes. Specifically, every time, we randomly choose one node out of N RPC servers to request 10 flows serially, and we repeat this $2,000,000$ times. Figure 6 shows the 99th and 99.9th percentile values of the workflow completion time, with an increasing number of bad nodes among a total of 100 instances.

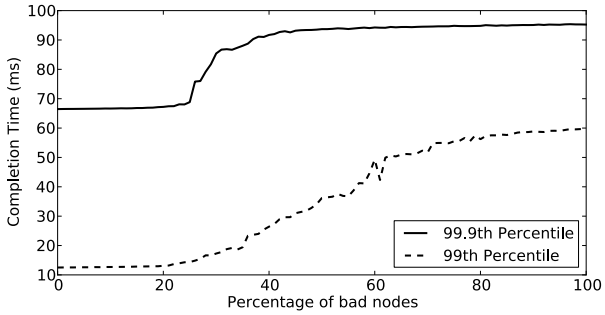


Figure 6: Impact of bad nodes on the flow tail completion times of the sequential model. Bobtail can expect to reduce tail flow completion time even when as many as 20% of nodes are bad.

Interestingly, there is no difference in the tails of overall completion times when as many as 20% of nodes are bad. But the difference in flow tail completion time between 20% bad nodes and 50% bad nodes is severe: flow completion time increases by *a factor of three* at the 99th percentile, and a similar pattern exists at the 99.9th percentile with a smaller difference. This means Bobtail is allowed to make mistakes—even if up to 20% of the instances picked by Bobtail are actually bad VMs, it still helps reduce flow completion time when compared to using random instances from EC2. Our measurements suggest that receiving 50% bad nodes from EC2 is not uncommon.

Figure 7 shows the completion time of the partition-aggregation model when there are 10, 20, and 40 nodes in the workloads. At modest scales, with fan-outs of 10 or even 20 nodes, there are substantial gains to be realized by avoiding bad nodes. However, there is less room for error here than in the sequential model: as the system scales up, other barriers present themselves, and avoiding nodes we classify as bad provides diminishing returns. Understanding this open question is an important challenge for us going forward.

4.2 System Design and Implementation

Bobtail needs to be a scalable system that makes accurate decisions in a timely fashion. While the node property remains stable in our five-week measurement, empirical evidence shows that the longer Bobtail runs, the more accurate its result can be. However, because launching an instance takes no more than a minute in EC2, we limit Bobtail to making a decision in under two minutes. Therefore, we need to strike a balance between accuracy and scalability.

A naive approach might be to simply conduct network measurements with every candidate. But however accurate it might be, such a design would not scale well to handle a large number of candidate instances in parallel: to do so in a short period of time would require sending a large amount of network traffic as quickly as possible to all candidates, and the synchronous nature of the measurement could cause severe network congestion or even TCP incast [23].

On the other hand, the most scalable approach involves conducting testing locally at the candidate instances, which does not rely on any resources outside the instance itself. Therefore, all operations can be done quickly and in parallel. This approach trades accuracy for scalability. Fortunately, Figures 6 and 7 show that Bobtail is allowed to make mistakes.

Based on our root cause analysis, such a method exists because the part of the long tail problem we focus on is *a property of nodes instead of the network*. Accordingly, if we know the workload patterns of the VMs co-located with the victim VM, we should be able to predict if the victim VM will have a bad latency distribution locally without any network measurement.

In order to achieve this, we must infer how often long scheduling delays happen to the victim VM. Because the long scheduling delays caused by the co-located CPU-intensive VMs are not unique to network packet processing and any interrupt-based events will suffer from the same problem, we can measure the frequency of large delays by measuring the time for the target VM to wake up from the `sleep` function call—the delay to process the timer interrupt is a proxy for delays in processing all hardware interrupts.

To verify this hypothesis, we repeat the five controlled experiments presented in the root cause analysis. But instead of running an RPC server in the victim VM and measuring the RTTs with another client, the victim VM runs a program that loops to sleep 1*ms* and measures the *wall time* for the `sleep` operation. Normally, the VM should be able to wake up after a little over 1*ms*, but co-located CPU-intensive VMs may prevent it from doing so, which results in large delays.

Figure 8 shows the number of times when the sleep time rises above 10*ms* in the five scenarios of the controlled experiments. As expected, when two or more VMs are CPU-intensive, the number of large delays experienced by the victim VM is *one to two orders of magnitude* above that experienced when zero or one VMs are CPU-intensive. Although the fraction of such large delays is small in all scenarios, the large difference in the raw counts forms a clear criterion for distinguishing bad nodes from good nodes. In addition, although it is not shown in the figure, we find that large delays with zero or one CPU-intensive VMs mostly appear for lengths of around 60*ms* or 90*ms*; these are caused by the 40% CPU cap on each latency-sensitive VM (i.e., when they are not allowed to use the CPU despite its availability). Delays

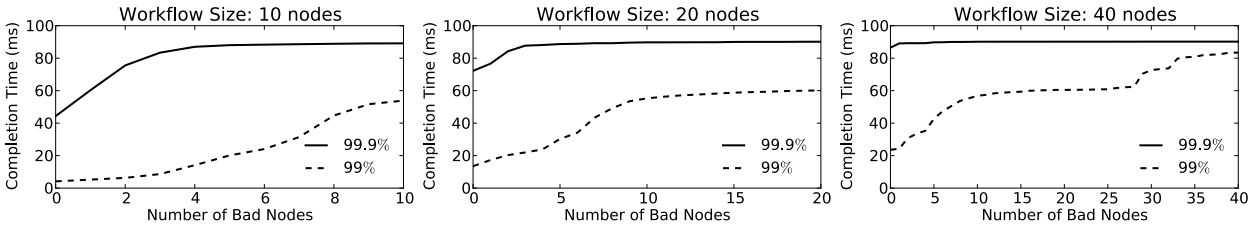


Figure 7: Impact of bad nodes on the tail completion time of the partition-aggregation model with 10, 20, and 40 nodes involved in the workloads. At modest scales, with fan-outs of 10 or even 20 nodes, there are substantial gains to be realized by avoiding bad nodes.

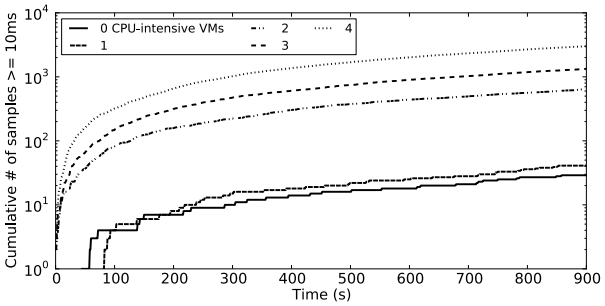


Figure 8: The number of large scheduling delays experienced by the victim VM in controlled experiments with an increasing number of VMs running CPU-intensive workloads. Such large delay counts form a clear criterion for distinguishing bad nodes from good nodes.

Algorithm 1 Instance Selection Algorithm

```
1:  num_delay = 0
2:  for i = 1 → M do
3:      sleep for S micro seconds
4:      if sleep time ≥ 10ms then
5:          num_delay++
6:      end if
7:  end for
8:  if num_delay ≤ LOW_MARK then
9:      return GOOD
10: end if
11: if num_delay ≤ HIGH_MARK then
12:     return MAY USE NETWORK TEST
13: end if
14: return BAD
```

experienced in other scenarios are more likely to be below 30*ms*, which is a result of latency-sensitive VMs preempting CPU-intensive VMs. This observation can serve as another clue for distinguishing the two cases.

Based on the results of our controlled experiments, we can design an instance selection algorithm to predict *locally* if a target VM will experience a large number of long scheduling delays. Algorithm 1 shows the pseudocode of our design. While the algorithm itself is straightforward, the challenge is to find the right threshold in EC2 to distinguish the two cases (`LOW_MARK` and `HIGH_MARK`) and to draw an accurate conclusion as quickly as possible (loop size M).

Our current policy is to reduce false positives, because in the partition-aggregation pattern, reducing bad nodes is critical to scalability. The cost of such conservatism is that we may label good nodes as bad incorrectly, and as a result we must instantiate even more nodes to reach a desired number. To return N good nodes as requested by users, our system needs to launch $K * N$ instances, and then it needs to find the best N instances of that set with the lowest probability of producing long latency tails.

After Bobtail fulfills a user's request for N instances whose delays fall below `LOW_MARK`, we can apply the network-based latency testing to the leftover instances whose delays fall between `LOW_MARK` and `HIGH_MARK`; this costs the user nothing but provides further value using the instances that users already paid for by the hour. Many of these nodes are likely false negatives which, upon further inspection, can be approved and returned to the user. In this scenario, scalability is no longer a problem because we no longer need to make a decision within minutes. Aggregate network throughput for testing can be thus much reduced. With this optimization, we may achieve a much lower effective false negative rate, which will be discussed in the next subsection.

A remaining question is what happens if users run latency-sensitive workloads on the good instances Bobtail picked, but those VMs become bad after some length of time. In practice, because users are running network workloads on these VMs, they can tell if any good VM turns bad by inspecting their application logs without any extra monitoring effort. If it happens, users may use Bobtail to pick more good VMs to take the place of the bad ones. Fortunately, as indicated in Figure 3, our five-week measurement shows that such properties generally persist, so workload migration does not need to happen very frequently. In addition, Figures 6 and 7 also indicate that

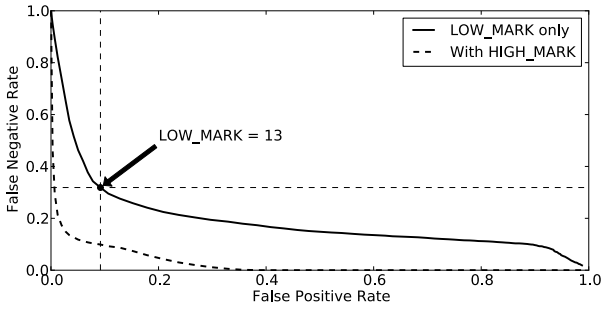


Figure 9: Trade-off between false positive and false negative rates of the instance selection algorithm. Our system can achieve a < 0.1 false positive rate while maintaining a false negative rate of around 0.3. With the help of network-based testing, the effective false negative rate can be reduced to below 0.1.

even if 20% of instances running latency-sensitive workloads are bad VMs, their impact on the latency distribution of sequential or partition-aggregation workloads is limited.

4.3 Parameterization

To implement Bobtail's algorithm, we need to define both its runtime (loop size M) and the thresholds for the `LOW_MARK` and `HIGH_MARK` parameters. Our design intends to limit testing time to under two minutes, so in our current implementation we set the loop size M to be $600K$ `sleep` operations, which translates to about 100 seconds on small instances in EC2—the worse the instance is, the longer it takes.

The remaining challenge we face is finding the right thresholds for our parameters (`LOW_MARK` and `HIGH_MARK`). To answer this inquiry, we launch 200 small instances from multiple availability zones (AZs) in EC2's US east region, and we run the selection algorithm for an hour on all the candidates. Meanwhile, we use the results of network-based measurements as the *ground truth* of whether the candidates are good or bad. Specifically, we consider the instances with 99.9th percentiles under 10*ms* for *all* micro benchmarks, which are discussed in § 5.1, as good nodes; all other nodes are considered bad.

Figure 9 shows the trade-off between the false positive and false negative rates by increasing `LOW_MARK` from 0 to 100. The turning point of the solid line appears when we set `LOW_MARK` around 13, which lets Bobtail achieve a < 0.1 false positive rate while maintaining a false negative rate of around 0.3—a good balance between false positive and false negative rates. Once `HIGH_MARK` is introduced (as five times `LOW_MARK`), the effective false negative rate can be reduced to below 0.1, albeit with the help of network-based testing. We leave it as future work to study when we need to re-calibrate these parameters.

The above result reflects our principle of favoring a low false positive. Therefore, we need to use a relatively large K value in order to get N good nodes from $K*N$ candidates. Recall that our measured good node ratio for random instances directly returned by EC2 ranges from 0.4 to 0.7. Thus, as an estimation, with a 0.3 false negative rate and a 0.4 to 0.7 good node ratio for random instances from multiple data centers, we need $K*N*(1-0.3)*0.4 = N$ or $K \approx 3.6$ to retrieve the number of desired good nodes from one batch of candidates. However, due to the pervasiveness of bad instances in EC2, even if Bobtail makes no mistakes we still need a minimum of $K*N*0.4 = N$ or $K = 2.5$. If startup latency is the critical resource, rather than the fees paid to start new instances, one can increase this factor to improve response time.

5 Evaluation

In this section, we evaluate our system over two availability zones (AZs) in EC2's US east region. These two AZs always return some bad nodes. We compare the latency tails of instances both selected by our system and launched directly via the standard mechanism. We conduct this comparison using both micro benchmarks and models of sequential and partition-aggregation workloads.

In each trial, we compare 40 small instances launched directly by EC2 from one AZ to 40 small instances selected by our system from the same AZ. The comparison is done with a series of benchmarks; these small instances will run RPC servers for all benchmarks. To launch 40 good instances, we use $K = 4$ with 160 candidate instances. In addition, we launch four extra large instances for every 40 small instances to run RPC clients. We do this because, as discussed earlier, extra large instances do not experience the extra long tail problem; we therefore can blame the server instances for bad latency distributions.

5.1 Micro Benchmarks

Our traffic models for both micro benchmarks and sequential and partition-aggregation workloads have inter-arrival times of RPC calls forming a Poisson process. For micro benchmarks, we assign 10 small instance servers to each extra large client. The RPC call rates are set at 100, 200, and 500 calls/second. In each RPC call, the client sends an 8-byte request to the server, and the server responds with 2KB of random data. Meanwhile, both requests and responses are packaged with another 29-byte overhead. The 2KB message size was chosen because measurements taken in a dedicated data center in-

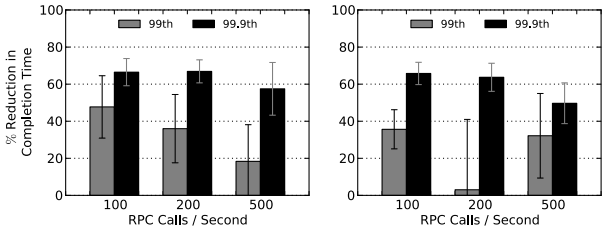


Figure 10: Reduction in flow tail completion time in micro benchmarks by using Bobtail in two availability zones in EC2's US east region. The mean reduction time is presented with a 90% confidence interval.

dicate that most latency-sensitive flows are around 2KB in size [1]. Note that we do not generate artificial background traffic, because real background traffic already exists throughout EC2 where we evaluate Bobtail.

Figure 10 presents the reductions in completion times for three RPC request rates in micro benchmarks across two AZs. Bobtail reduces latency at the 99.9th percentile from 50% to 65%. In micro benchmark and subsequent evaluations, the mean of reduction percentages in flow completion is presented with a 90% confidence interval.

However, improvements at the 99th percentile are smaller with a higher variance. This is because, as shown in Figure 1, the 99th percentile RTTs within EC2 are not very bad to begin with ($\sim$2.5ms); therefore, Bobtail's improvement space is much smaller at the 99th percentile than at the 99.9th percentile. For the same reason, network congestion may have a large impact on the 99th percentile while having little impact on the 99.9th percentile in EC2. The outlier of 200 calls/second in the second AZ of Figure 10 is caused by one trial in the experiment with 10 good small instances that exhibited abnormally large values at the 99th percentile.

5.2 Sequential Model

For sequential workloads, we apply the workload model to 20-node and 40-node client groups, in addition to the 10-node version shown in the micro benchmarks. In this case, the client sends the same request as before, but the servers reply with a message size randomly chosen from among 1KB, 2KB, and 4KB. For each workflow, instead of sending requests to all the servers, the client will randomly choose one server from the groups of sizes 10, 20, and 40. Then, it will send 10 synchronous RPC calls to the chosen server; the total time to complete all 10 RPC requests is then used as the workflow RTT. Because of this, the workflow rates for the sequential model are reduced to one tenth of the RPC request rates for micro benchmarks and become 10, 20, and 50 workflows per second.

Figure 11 shows our improvement under the sequential model with different numbers of RPC servers involved. Bobtail brings a 35% to 40% improvement to sequential workloads at the 99th percentile across all experiments, and it roughly translates to an 8ms reduction. The lengths of the confidence intervals grow as the number of server nodes increases; this is caused by a relatively smaller sample space. The similarity in the reduction of flow completion time with different numbers of server nodes shows that the tail performance of the sequential workflow model only depends on the ratio of bad nodes among all involved server nodes. Essentially, the sequential model demonstrates the average tail performance across all server nodes by randomly choosing one server node each time with equal probability at the client side.

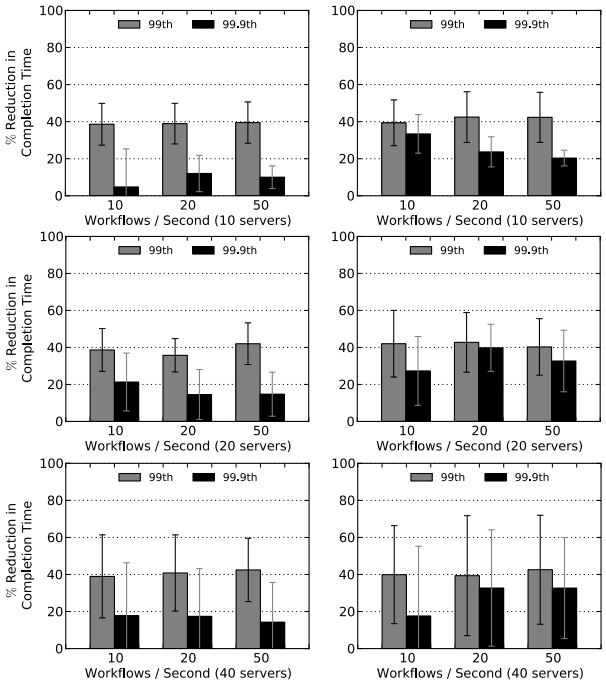


Figure 11: Reduction in flow tail completion time for sequential workflows by using Bobtail in two availability zones in EC2's US east region. The mean reduction time is presented with a 90% confidence interval.

Interestingly, and unlike in the micro benchmarks, improvement at the 99.9th percentile now becomes smaller and more variable. However, this phenomenon does match our simulation result shown in Figure 6 when discussing the potential benefits of using Bobtail.

5.3 Partition-Aggregation Model

For the partition-aggregation model, we use the same 10, 20, and 40-node groups to evaluate Bobtail. In this case, the client always sends requests to all servers in the group concurrently, and the workflow finishes once the slowest

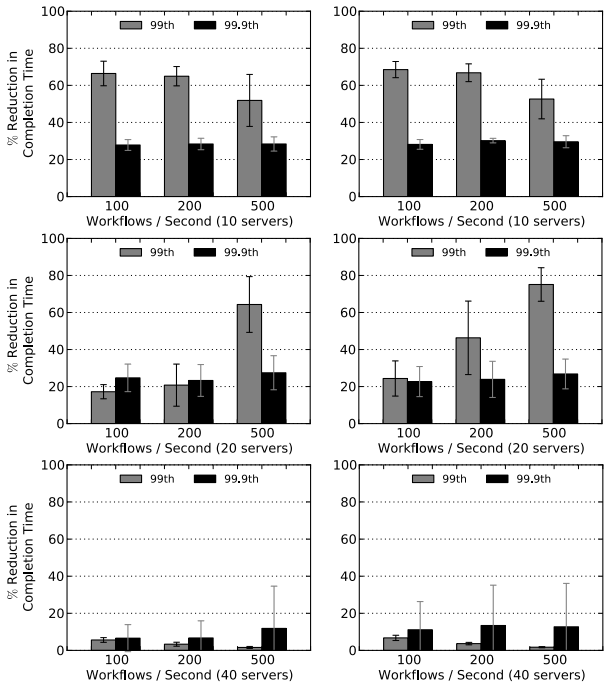


Figure 12: Reduction in flow tail completion time for partition-aggregation workflows by using Bobtail in two availability zones in EC2's US east region. The mean reduction time is presented with a 90% confidence interval.

RPC response returns; servers always reply with 2KB of random data. In other words, the RTT of the slowest RPC call is effectively the RTT of the workflow. Meanwhile, we keep the same workflow request rate from the micro benchmarks.

Figure 12 shows improvement under the partition-aggregation model with different numbers of RPC servers involved. Bobtail brings improvement of 50% to 65% at the 99th percentile with 10 servers. Similarly to the sequential workloads, the improvement at the 99.9th percentile is relatively small. In addition, as predicted by Figure 7, the reduction in tail completion time diminishes as the number of servers involved in the workload increases. To fully understand this phenomenon, we need to compare the behaviors of these two workload models.

For sequential workloads with random worker assignment, a small number of long-tail nodes have a modest impact. Intuitively, each such node has a $1/N$ chance of being selected for any work item and may (or may not) exhibit long tail behavior for that item. However, when this does happen, the impact is non-trivial, as the long delays consume the equivalent of many "regular" response times. So, one must minimize the pool of long-tail nodes in such architectures but needs not to avoid them entirely.

The situation is less pleasant for parallel, scatter-gather style workloads. In such workloads, long-tail nodes act as the barrier to scalability. Even a relatively low percentage of long-tail nodes will cause significant slowdowns overall, as each phase of the computation runs at the speed of the slowest worker. Reducing or even eliminating long-tail nodes removes this early barrier to scale. However, it is not a panacea. As the computation fans out to more nodes, other limiting factors come into play, reducing the effectiveness of further parallelization. We leave it as future work to study other factors that cause the latency tail problem with larger fan-out in cloud data centers.

6 Discussion

Emergent partitions A naive interpretation of Bobtail's design is that a given customer of EC2 simply seeks out those nodes which have at most one VM per CPU. If this were the case, deploying Bobtail widely would result in a race to the bottom. However, not all forms of sharing are bad. Co-locating multiple VMs running latency-sensitive workloads would not give rise to the scheduling anomaly at the root of our problem. Indeed, wide deployment of Bobtail for latency-sensitive jobs would lead to placements on nodes which are either under-subscribed or dominated by other latency-sensitive workloads. Surprisingly, this provides value to CPU-bound workloads as well. Latency-sensitive workloads will cause frequent context switches and reductions in cache efficiency; both of these degrade CPU-bound workload performance. Therefore, as the usage of Bobtail increases in a cloud data center, we expect it will eventually result in emergent partitions: regions with mostly CPU-bound VMs and regions with mostly latency-sensitive VMs. However, to validate this hypothesis, we would need direct access to the low-level workload characterization of cloud data centers like EC2.

Alternative solutions Bobtail provides a user-centric solution that cloud users can apply today to avoid long latency tails without changing any of the underlying infrastructure. Alternatively, cloud providers can offer their solutions by modifying the cloud infrastructure and placement policy. For example, they can avoid allocating more than C VMs on a physical machine with C processors, at the cost of resource utilization. They can also overhaul their VM placement policy to allocate different types of VMs in different regions in the first place. In addition, new versions of the credit scheduler [28] may also help alleviate the problem.

7 Related Work

Latency in the Long Tail Proposals to reduce network latency in data centers fall into two broad cate-

gories: those that reduce network congestion and those that prioritize flows according to their latency sensitivity. Alizadeh *et al.* proposed to reduce switch buffer occupancy time by leveraging Explicit Congestion Notification (ECN) to indicate the degree of network congestion rather than whether congestion exists [1]. Follow-up work further reduced the buffer occupancy time by slightly capping bandwidth capacity [2]. Wilson *et al.* and Vamanan *et al.* both argued that the TCP congestion control protocols used in data centers should be deadline-aware [26, 22]. Hong *et al.* designed a flow scheduling system for data centers to prioritize latency-sensitive jobs with flow preemption [10]. Zats *et al.* proposed a cross-stack solution that combined ECN with application-specified flow priorities and adaptive load balancing in an effort to unify otherwise disparate knowledge about the state of network traffic [29].

The above solutions focus on the component of long tail flow completion times that is the result of the network alone and, as such, are complementary to our approach. We have shown that the co-scheduling of CPU-intensive and latency-sensitive workloads in virtualized data centers can result in a significant increase in the size of the long tail, and that this component of the tail can be addressed independently of the network.

The Xen Hypervisior and its Scheduler In § 3, we discussed how Xen uses a credit-based scheduler [28] that is not friendly to latency-sensitive workloads. Various characteristics of this credit scheduler have been examined, including scheduler configurations [15], performance interference caused by different types of co-locating workloads [15, 12, 25], and the source of overhead incurred by virtualization on the network layer [25]. Several designs have been proposed to improve the current credit scheduler, and they all share the approach of boosting the priority of latency-sensitive VMs while still maintaining CPU fairness in the long term [9, 8, 11]. However, the degree to which such approaches will impact the long tail problem at scale has yet to be studied.

Instead of improving the VM scheduler itself, Wood *et al.* created a framework for the automatic migration of virtual machines between physical hosts in Xen when resources become a bottleneck [27]. Mei *et al.* also pointed out that a strategic co-placement of different workload types in a virtualized data center will improve performance for both cloud consumers and cloud providers [14]. Our work adopts a similar goal of improving the tail completion time of latency-sensitive workloads for individual users while also increasing the overall efficiency of resource usage across the entire virtualized data center. However, our solution does not require the collaboration of cloud providers, and many cloud customers can deploy our system independently.

EC2 Measurements Wang *et al.* showed that the network performance of EC2 is much more variable than that of non-virtualized clusters due to virtualization and processor sharing [24]. In addition, Schad *et al.* found a bimodal performance distribution with high variance for most of their metrics related to CPU, disk I/O, and network [19]. Barker *et al.* also quantified the jitter of CPU, disk, and network performance in EC2 and its impact on latency-sensitive applications [4]. Moreover, A. Li *et al.* compared multiple cloud providers, including EC2, using many types of workloads and claimed that there is no single winner on all metrics [13]. These studies only investigate the average and variance of their performance metrics, while the focus of our study is on the tail of network latency distributions in EC2.

Ou *et al.* considered hardware heterogeneity within EC2, and they noted that within a single instance type and availability zone, the variation in performance for CPU-intensive workloads can be as high as 60% [16]. They made clear that one easy way to improve instance performance is to check the model of processor assigned. While selecting instances also represents the core of our work, Bobtail examines dynamic properties of EC2 as opposed to static configuration properties.

8 Conclusion

In this paper, we demonstrate that virtualization used in EC2 exacerbates the long tail problem of network round-trip-times by a factor of two to four. Notably, we find that poor response times in the cloud are a property of nodes rather than the network, and that the long latency tail problem is pervasive throughout EC2 and persistent over time. Using controlled experiments, we show that co-scheduling of CPU-bound and latency-sensitive tasks causes this problem. We present a system, Bobtail, which proactively detects and avoids these bad neighboring VMs without significantly penalizing node instantiation. Evaluations in two availability zones in EC2's US east region show that common communication patterns benefit from reductions of up to 40% in their 99.9th percentile response times.

9 Acknowledgments

We are grateful to the anonymous reviewers and our shepherd, George Porter, for their comments on this paper. This work was supported in part by the Department of Homeland Security (DHS) under contract numbers D08PC75388, and FA8750-12-2-0314, the National Science Foundation (NSF) under contract numbers CNS 1111699, CNS 091639, CNS 08311174, and CNS 0751116, and the Department of the Navy under contract N000.14-09-1-1042.

References

[1] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 conference (SIGCOMM'10)* (New Delhi, India, August 2010).

[2] ALIZADEH, M., KABBANI, A., EDSALL, T., PRABHAKAR, B., VAHDAT, A., AND YASUDA, M. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12)* (San Jose, CA, USA, April 2012).

[3] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)* (Bolton Landing, NY, USA, October 2003).

[4] BARKER, S. K., AND SHENOY, P. Empirical Evaluation of Latency-sensitive Application Performance in the Cloud. In *Proceedings of the 1st annual ACM SIGMM conference on Multimedia systems (MMSys'10)* (Scottsdale, AZ, USA, February 2010).

[5] BOUCH, A., KUCHINSKY, A., AND BHATTI, N. Quality is in the eye of the beholder: meeting users' requirements for Internet quality of service. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI'00)* (The Hague, The Netherlands, April 2000).

[6] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI'04)* (San Francisco, CA, USA, March 2004).

[7] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)* (Stevenson, WA, USA, October 2007).

[8] DUNLAP, G. W. Scheduler Development Update. In *Xen Summit Asia 2009* (Shanghai, China, November 2009).

[9] GOVINDAN, S., NATH, A. R., DAS, A., URGAONKAR, B., AND SIVASUBRAMANIAM, A. Xen and Co.: Communication-Aware CPU Scheduling for Consolidated Xen-based Hosting Platforms. In *Proceedings of the 3rd international conference on Virtual execution environments (VEE'07)* (San Diego, CA, 2007, June 2007).

[10] HONG, C.-Y., CAESAR, M., AND GODFREY, P. B. Finishing Flows Quickly with Preemptive Scheduling. In *Proceedings of the ACM SIGCOMM 2012 conference (SIGCOMM'12)* (Helsinki, Finland, August 2012).

[11] KIM, H., LIM, H., JEONG, J., JO, H., AND LEE, J. Task-aware Virtual Machine Scheduling for I/O Performance. In *Proceedings of the 5th international conference on virtual execution environments (VEE'09)* (Washington, DC, USA, March 2009).

[12] KOH, Y., KNAUERHASE, R. C., BRETT, P., BOWMAN, M., WEN, Z., AND PU, C. An Analysis of Performance Interference Effects in Virtual Environments. In *Proceedings of the 2007 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'07)* (San Jose, CA, USA, April 2007).

[13] LI, A., YANG, X., KANDULA, S., AND ZHANG, M. CloudCmp: Comparing Public Cloud Providers. In *Proceedings of the 2010 Internet Measurement Conference (IMC'10)* (Melbourne, Australia, November 2010).

[14] MEI, Y., LIU, L., PU, X., AND SIVATHANU, S. Performance Measurements and Analysis of Network I/O Applications in Virtualized Cloud. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing (CLOUD'10)* (Miami, FL, June 2010).

[15] ONGARO, D., COX, A. L., AND RIXNER, S. Scheduling I/O in virtual machine monitors. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'09)* (Washington, DC, USA, March 2008).

[16] OU, Z., ZHUANG, H., NURMINEN, J. K., YLÄ-JÄÄSKI, A., AND HUI, P. Exploiting Hardware Heterogeneity within the Same Instance Type of Amazon EC2. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'12)* (Boston, MA, USA, June 2012).

[17] PATTERSON, D. A. Latency lags bandwidth. *Communication of ACM 47*, 10 (Oct 2004), 71–75.

[18] RUMBLE, S. M., ONGARO, D., STUTSMAN, R., ROSENBLUM, M., AND OUSTERHOUT, J. K. It's Time for Low Latency. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS XIII)* (Napa, CA, USA, May 2011).

[19] SCHAD, J., DITTRICH, J., AND QUIANÉ-RUIZ, J.-A. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. In *Proceedings of the 36th International Conference on Very Large Data Bases (VLDB'10)* (Singapore, September 2010).

[20] SLEE, M., AGARWAL, A., AND KWIATKOWSKI, M. Thrift: Scalable Cross-Language Services Implementation. Tech. rep., Facebook, Palo Alto, CA, USA, April 2007.

[21] TECHCRUNCH. There Goes The Weekend! Pinterest, Instagram And Netflix Down Due To AWS Outage. http://techcrunch.com/2012/06/30/there-goes-the-weekend-pinterest-instagram-and-netflix-down-due-to-aws-outage/.

[22] VAMANAN, B., HASAN, J., AND VIJAYKUMAR, T. N. Deadline-Aware Datacenter TCP (D^2TCP). In *Proceedings of the ACM SIGCOMM 2012 conference (SIGCOMM'12)* (Helsinki, Finland, August 2012).

[23] VASUDEVAN, V., PHANISHAYEE, A., SHAH, H., KREVAT, E., ANDERSEN, D. G., GANGER, G. R., GIBSON, G. A., AND MUELLER, B. Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication. In *Proceedings of the ACM SIGCOMM 2009 conference (SIGCOMM'09)* (Barcelona, Spain, August 2009).

[24] WANG, G., AND NG, T. S. E. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *Proceedings of the 29th conference on Information communications (INFOCOM'10)* (San Diego, CA, USA, March 2010).

[25] WHITEAKER, J., SCHNEIDER, F., AND TEIXEIRA, R. Explaining Packet Delays Under Virtualization. *SIGCOMM Computer Communication Review 41*, 1 (January 2011), 38–44.

[26] WILSON, C., BALLANI, H., KARAGIANNIS, T., AND ROWTRON, A. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 conference (SIGCOMM'11)* (Toronto, ON, CA, August 2011).

[27] WOOD, T., SHENOY, P. J., VENKATARAMANI, A., AND YOUSIF, M. S. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proceedings of the 4th conference on Symposium on Networked Systems Design & Implementation (NSDI'07)* (Cambridge, MA, USA, April 2007).

[28] XEN.ORG. Xen Credit Scheduler. `http://wiki.xen.org/wiki/Credit_Scheduler`.

[29] ZATS, D., DAS, T., MOHAN, P., BORTHAKUR, D., AND KATZ, R. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2012 conference (SIGCOMM'12)* (Helsinki, Finland, August 2012).

Rhea: automatic filtering for unstructured cloud storage

Christos Gkantsidis, Dimitrios Vytiniotis, Orion Hodson
Dushyanth Narayanan, Florin Dinu, Antony Rowstron*
Microsoft Research, Cambridge, UK

Abstract

Unstructured storage and data processing using platforms such as MapReduce are increasingly popular for their simplicity, scalability, and flexibility. Using elastic cloud storage and computation makes them even more attractive. However cloud providers such as Amazon and Windows Azure separate their storage and compute resources even within the same data center. Transferring data from storage to compute thus uses core data center network bandwidth, which is scarce and oversubscribed. As the data is unstructured, the infrastructure cannot automatically apply selection, projection, or other filtering predicates at the storage layer. The problem is even worse if customers want to use compute resources on one provider but use data stored with other provider(s). The bottleneck is now the WAN link which impacts performance but also incurs egress bandwidth charges.

This paper presents Rhea, a system to automatically generate and run storage-side data filters for unstructured and semi-structured data. It uses static analysis of application code to generate filters that are safe, stateless, side effect free, best effort, and transparent to both storage and compute layers. Filters never remove data that is used by the computation. Our evaluation shows that Rhea filters achieve a reduction in data transfer of 2x–20,000x, which reduces job run times by up to 5x and dollar costs for cross-cloud computations by up to 13x.

1 Introduction

The last decade has seen a huge increase in the use of "noSQL" approaches to data analytics. Whereas in the past the default data store was a relational one (e.g. SQL), today it is possible and often desirable to store the data as unstructured files (e.g. text-based logs) and to process them using general-purpose languages (Java, C#). The combination of unstructured storage and general-purpose programming languages increases flexibility: different programs can interpret the same data in different ways, and changes in format can be handled by changing the code rather than restructuring the data.

This flexibility comes at a cost. The structure of the data is now *implicit* in the program code. Most analytics jobs use a subset of the input data, i.e. only some of the data items are relevant and only some of the fields within those are relevant. Since these selection and projection operations are embedded in the application code, they cannot be applied by the storage layer; rather all the data must be read into the application code.

This is not an issue for dedicated data processing infrastructures where a single cluster provides both storage and computation, and a framework such as MapReduce, Hadoop, or Dryad co-locates computation with data. However it is a problem when running such frameworks in an elastic cloud. Cloud providers such as Amazon and Windows Azure provide both scalable unstructured storage and elastic compute resources but these are physically disjoint. There are many good reasons for this including security, performance isolation, and the need to independently scale and provision the storage and elastic compute infrastructures. Both Amazon's S3 [1] and Windows Azure Storage [4, 39] follow this model of physically separate compute and storage servers within the same data center. This means that bytes transferred from storage to compute use core data center network bandwidth, which is often scarce and oversubscribed [14] (see also Section 4.1.1).

Our aim is to retain the flexibility of unstructured storage and the elasticity of cloud storage and computation, yet reduce the bandwidth costs of transferring redundant or irrelevant data from storage to computation. Specifically, we wish to transparently run applications written for frameworks such as Hadoop in the cloud, but extract the implicit structure and use it to reduce the amount of data read over the data center network. Reducing bandwidth will improve provider utilization, by allowing more jobs to be run on the same servers, and improve performance for customers, as their jobs will run faster.

*Work done while on internship from Rice University

Our approach is to use static analysis on application code to automatically generate application-specific *filters* that remove data that is irrelevant to the computation. The generated filters are then run (typically, but not necessarily) on storage servers in order to reduce bandwidth. Filters need to be safe and transparent to the application code. They need to be conservative, i.e., the output of the computation must be the same whether using filters or not, and hence only data that provably cannot affect the computation can be suppressed. Since filters are using spare computational resources on the storage servers, they also need to be best-effort, i.e. they can be disabled at any time without affecting the application.

Our *Rhea* system automatically generates and executes storage-side filters for unstructured text data. Rhea extracts both *row filters* which select out irrelevant rows (lines) in the input, as well as *column filters* which project out irrelevant columns (substrings) in the surviving rows.[1] Both row and column filters are safe, transparent, conservative, and best-effort. Rhea analyzes the Java bytecode of programs written for Hadoop MapReduce, producing job-specific executable filters.

Section 2 makes the case for implicit, storage-side filtering and describes 9 analytic jobs that we use to motivate and evaluate Rhea. Section 3 describes the design and implementation of Rhea and its filter generation algorithms. Section 4 shows that storage-to-compute bandwidth is scarce in real cloud platforms; that Rhea filters achieve substantial reduction in the storage-to-compute data transfer and that this leads to performance improvements in a cloud environment. Rhea reduces storage-to-compute traffic by a factor of 2–20,000, job run times by a factor of up to 5, and dollar costs for cross-cloud computations by a factor of up to 13. Section 5 discusses related work, and Section 6 concludes.

2 Background and Motivation

In this section we first describe the design rationale for Rhea: the network bottlenecks that motivate storage-side filtering, and the case for automatically generated (implicit) filters. We finally describe the example jobs that we use to evaluate Rhea.

2.1 Storage-side filtering

The case for storage-side filtering is based on two observations. First, compute cycles on storage servers are cheap relative to core network bandwidth. Of course, since this is not an explicitly provisioned resource, use of such cycles should be opportunistic and best-effort. Second, storage-to-compute bandwidth is a scarce resource that can be a performance bottleneck. Our measurements of read bandwidth for Amazon EC2/S3 and Windows Azure confirm this (Section 4.1.1) and are consistent with earlier measurements [11, 12].

If data must be transferred across data centers or availability zones, then this will not only use WAN bandwidth and impact performance, but also incur egress bandwidth charges for the user. This can happen if data stored in different geographical locations need to be combined, e.g., web logs from East and West Coast servers. Some jobs may need to combine public and private data, e.g. a public data set stored in Amazon S3 [31] with a private one stored on-premises, or a data set stored in Amazon S3 with one stored in Windows Azure Storage.

Our aim is to reduce network load, job run times, and egress bandwidth charges through filtering for many different scenarios. When the storage is in the cloud, the cloud provider (e.g. Amazon S3) could natively support execution of Rhea filters on or near the storage servers. In the case where the computation uses a compute cluster provided by the same provider (e.g. Amazon EC2 in the case of Amazon S3), the provider could even extract and deploy filters transparently to the customer. For on-premises ("private cloud") storage, filters could be deployed by the customer on the storage servers or near them, e.g. on the same rack. If the provider does not support filtering at the storage servers, filtering can still be used to reduce WAN data transfers by running the filters in a compute instance located in the same data center as the storage. In the latter case our evaluation shows that the savings in egress bandwidth charges outweigh the dollar cost of a filtering VM instance. Additionally, the isolation properties of Rhea filters make it possible for multiple users to safely share a single filtering VM and thus reduce this cost.

2.2 Implicit filtering

Rhea creates filters implicitly and transparently using static analysis of the programs. An alternative would be to have the programmer do this explicitly. For example a language like SQL makes the filtering predicates and columns accessed within each row explicit. E.g., the "WHERE" clause in a SQL statement identifies the filtering predicate and the "SELECT" statement for column selectivity. Several storage systems support explicit column selectivity for MapReduce jobs, e.g. "slice predicates" in Cassandra [3], "input format classes" in Zebra [41], explicit filters in Pig/Latin [13], and RC-files in Hive [34]. In such situations input data pre-filtering can be performed using standard techniques from database query optimization.

While extremely useful for this kind of query optimization and reasoning, explicit approaches often provide less flexibility, as the application is tied to a specific interface to the storage (SQL, Cassandra, etc). They are

[1] For convenience we use the term "row" to refer to the input unit of a Map process, and "column" to refer to the output of the tokenization performed on the row input according to some user-specified logic.

also less well-suited for free-format or semi-structured text files, which have to be parsed in an application-specific manner. This flexibility is one of the reasons that platforms such as SCOPE [5] allow a mixture of SQL-like and actual C# code. Eventually all code (including the SQL part) is compiled down to .NET and executed.

Our aim in Rhea is to handle the general case where programmers can embed application-specific column parsing logic or arbitrary code in the mapper, without imposing any additional programmer burden such as hand-annotating the code with filtering predicates. Instead, Rhea infers filters automatically using static analysis of the application byte code. Since Rhea only examines the application code, it is applicable even when *the format of the data is not known a-priori*, or *the data does not strictly conform to an input format* (for instance tabular input data with occasionally occurring comment lines starting with a special character).

2.3 Example analytics jobs

Our static analysis handles arbitrary Java byte code: we have used over 160 Hadoop mappers from various Hadoop libraries and other public and private sources to test our tool and validate the generated filters (Section 3.4). Of these, we present nine jobs for which data were also available and use them to drive our evaluation (Section 4). Here we describe these nine jobs. Note that we do not include commonly-used benchmarks such as Sort and WordCount, which are used to stress-test MapReduce infrastructures. Neither of these has any selectivity, i.e., the mapper examines all the input data, and thus Rhea would not generate any filters for them. However, we do not believe such benchmarks are representative of real-world jobs, which often do have selectivity.

GeoLocation This publicly available Hadoop example [24] groups Wikipedia articles by their geographical location. The input data is based on a publicly available data set [23]. The input format is text, with each line corresponding to a row and tab characters separating columns within the row. Each row contains a type column which determines how the rest of the row is interpreted; the example job only considers one of the two row types, and hence rows of the other type can be safely suppressed from the input.

Event log processing The next two jobs are based on processing event logs from a large compute/storage platform consisting of tens of thousands of servers. Users issue tasks to the system, which spawn processes on multiple servers. Resource usage information measured on all these servers is written to two event logs: a process log with one row per executed process, and an activity log that records fine-grained resource consumption information. We use two typical jobs that process this data. The first, *FindUserUsage*, identifies the top-k users by total

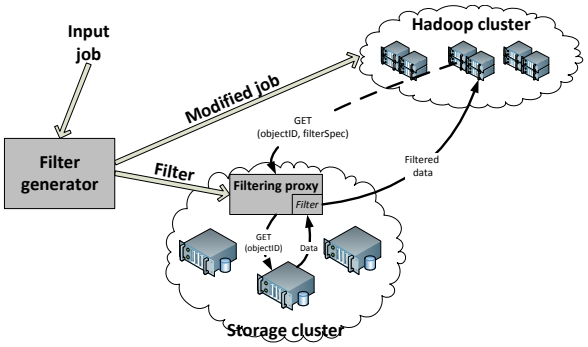


Figure 1: System architecture

process execution time. The second, *ComputeIoVolumes*, is a join: it filters out failed tasks by reading the process log and then computes storage I/O statistics for the successful tasks from the activity log.

IT log reporting The next job is based on enterprise IT logs across thousands of shared infrastructure machines on an enterprise network. The sample job (*IT Reporting*) queries these logs to find the aggregate CPU usage for a specific machine, grouped by the type of user generating the CPU load.

Web logs and ranking The last five jobs are from a benchmark developed by Pavlo et al. [30] to compare unstructured (MapReduce) and structured (DBMS) approaches to data analysis. The jobs all use synthetically generated data sets consisting of a set of HTML documents that link to each other, a Rankings table that maps each unique URL to its computed PageRank, and a UserVisits table that logs user visits to each unique URL as well as context information such as time-stamp, country code, ad revenue, and search context.

The first two jobs are variants of a *SelectionTask* (find all URLs with page rank higher than X). The amount of input data that is relevant to this task depends on the threshold X. Thus we use two variants with thresholds $X_{1\%}$ and $X_{10\%}$, where approximately 1% of the URLs have page rank higher than $X_{1\%}$, and 10% of the URLs have page rank higher than $X_{10\%}$. The next two jobs are based on an *AggregationTask*. They find total revenue grouped by unique source IP, and total revenue grouped by source network, respectively. Finally, the *JoinTask* finds the average PageRank of the pages visited by the source IP that generated the most revenue within a particular date range.

3 Design and Implementation

The current Rhea prototype is designed for Hadoop MapReduce jobs. It generates executable Java filters from the mapper class(es) for each job. It is important to note that although Rhea filters are executable, running a filter is different from running arbitrary applica-

tion code, for example running the entire mapper task on the storage server. Filters are guaranteed to be safe and side-effect free and thus can be run with minimal sandboxing, with multiple filters from different jobs or customers co-existing in the same address space. They are transparent and best-effort, and hence can be disabled at any point to save resources without affecting the application. They are stateless and do not consume large amounts of memory to hold output, as is done by many mappers. Finally, they are guaranteed never to output more data than input. This is not true of mappers where the output data can be larger than the input data [6].

Figure 1 shows the architecture of Rhea, which consists of two components: a *filter generator* and a *filtering proxy*. The filter generator creates the filters and uploads them to the filtering proxy, and also adds a transparent, application-independent, client-side shim to the user's Hadoop job to create a Rhea-aware version of the job. The Rhea-aware version of the job intercepts cloud storage requests, and redirects them to the filtering proxy. The redirected requests include a description of the filter to be instantiated and a serialized cloud storage REST request to access the job's data. The serialized request is signed with the user's cloud storage provider credentials when it is generated on the Hadoop nodes so the filtering proxy holds no confidential user state. When the filtering proxy receives the redirected request, it instantiates the required filter, issues the signed storage request, and returns filtered data to the caller. Thus Rhea filtering is transparent to the user code, to the elastic compute infrastructure, and to the storage layer, and requires no sensitive user state. The proxy works with Amazon's S3 Storage and Windows Azure Storage, and also has a local file system back end for development and test use.

The filter generator takes the Java byte code of a Hadoop job, and generates a *row filter* and a *column filter* for each mapper class found in the program. These are encoded as methods on an extension of the corresponding mapper class. The extended classes are shipped to the filtering proxy as Java jar files and dynamically loaded into its address space. The filter generator, and the static analysis underlying it, are implemented using SAWJA [18], a tool which provides a high-level stackless representation of Java byte code. In the rest of this section we describe the static analysis used for row and column filter generation.

3.1 Row Filters

A row filter in Rhea is a method that takes a single record as input and returns `false` if that record does not affect the result of the MapReduce computation, and `true` otherwise. It can have false positives, i.e., return `true` for records that do not affect the output, but it can not have false negatives. The byte code of the filter is generated from that of the mapper. Intuitively, it is a stripped-down or "skeleton" version of the mapper, retaining only those instructions and execution paths that determine whether or not a given invocation will produce an output. Instructions that are used to compute the *value* of the output but do not affect the control flow are not present in the filter. As such, the row filter is *completely independent of the format of the input data* and only depends on the predicates that the mapper is using on the input.

Listing 1 shows a typical example: the mapper for the GeoLocation job (Section 2.3). It tokenizes the input `value` (line 7), extracts the first three tokens, (line 9–11), and then checks if the second token equals the static field `GEO_RSS_URI` (line 13). If it does, more processing follows (line 14–26) and some value is output on `outputCollector`; otherwise, no output is generated.

```
1 ... // class and field declarations
2 public void map(LongWritable key, Text value,
3  OutputCollector<Text, Text> outputCollector,
4  Reporter reporter) throws IOException {
5
6  String dataRow = value.toString();
7  StringTokenizer dataTokenizer =
8      new StringTokenizer(dataRow, "\t");
9  String artName = dataTokenizer.nextToken();
10 String pointTyp = dataTokenizer.nextToken();
11 String geoPoint = dataTokenizer.nextToken();
12
13 if (GEO_RSS_URI.equals(pointTyp)) {
14   StringTokenizer st =
15       new StringTokenizer(geoPoint, "␣");
16   String strLat = st.nextToken();
17   String strLong = st.nextToken();
18   double lat = Double.parseDouble(strLat);
19   double lang = Double.parseDouble(strLong);
20   long roundedLat = Math.round(lat);
21   long roundedLong = Math.round(lang);
22   String locationKey = ...
23   String locationName = ...
24   locationName = ...
25   geoLocationKey.set(locationKey);
26   geoLocationName.set(locationName);
27   outputCollector.collect(geoLocationKey,
28             geoLocationName);
29 } }
```

Listing 1: GeoLocation map job

Listing 2 shows the filter generated by Rhea for this mapper. It also tokenizes the input (line 8) and performs the comparison on the second token (line 12) (`bcvar8` here corresponds to `pointTyp` in `map`). This test determines whether `map` would have produced output, and hence `filter` returns the corresponding Boolean value.

Comparison of `map` and `filter` reveals two interesting details. First, while `map` extracted three tokens from the input, `filter` only extracted two. The third token does not determine whether or not output is produced, although it does affect the value of the output. The static

```
1 public boolean filter (LongWritable bcvar1,
2    Text bcvar2,
3    OutputCollector bcvar3,
  Reporter bcvar4) {
4
5  boolean cond = false;
6  String bcvar5 = bcvar2.toString();
7  String irvar0 = "\t";
8  StringTokenizer bcvar6 =
9    new StringTokenizer(bcvar5,irvar0);
10 String bcvar7 = bcvar6.nextToken();
11 String bcvar8 = bcvar6.nextToken();
12 boolean irvar0_1=
13     GEO_RSS_URI.equals(bcvar8);
14
15 cond = ((irvar0_1?1:0)  !=  0);
16 if (!cond) return false;
17 return true;
18 }
```

Listing 2: Row filter generated for GeoLocation

analysis detects this and omits the extraction of the third token. Second, `map` does substantial processing (line 14–26) before producing the output. All these instructions are omitted from the filter: they affect the output value but not the output condition.

Row filter generation uses a variant of dependency analysis commonly found in program slicing [17, 26, 36]. Our analysis is based on the following steps:

1. It first identifies "output labels", i.e. program points at which the mapper produces output, such as calls to the Hadoop API including `OutputCollector.collect` (line 28 of Listing 1). The generated filter must return `true` for any input that causes the mapper to reach such an output label (line 17 of Listing 2). This basic definition of output label is later extended to handle the use of state in the mapper (Section 3.1.1).

2. The next step is to collect all control flow paths (including loops) of the mapper that reach an output label. Listing 1 contains a single control path that reaches an output label through line 13 of Listing 1.

3. Next, Rhea performs a label-flow analysis (as a standard forward analysis [29]) to compute a "flow map": for each program instruction, and for each variable referenced in that instruction, it computes all other labels that could affect the value of that variable.

4. For every path identified in Step 2, we keep only the instructions that, according to the flow map from Step 3, can affect any control flow decisions (line 6–13 of Listing 1, which correspond to line 6–16 of Listing 2). The result is a new set of paths which contains potentially fewer instructions per path – only the *necessary* ones for control flow to reach the path's output instruction.

5. Finally, we generate code for the *disjunction* of the paths computed in Step 4, emitting `return true` statements after the last conditional along each path. Technically, prior to this step we perform several optimizations, for instance we merge paths when both the `True` and the `False` case of a conditional statement can lead to output. We also never emit code for a loop if the continuation of a loop may reach an output instruction: in this case we simply `return true` when we reach the loop header, in order to avoid performing a potentially expensive computation if there is possibility of output after the loop.

3.1.1 Stateful mappers

This basic approach described above guarantees that the filter returns `true` for any input row for which the original mapper would produce output, but neglects the fact that `map` will be invoked on *multiple* rows, where each invocation may affect some *state* in the mapper that could affect the control flow in a subsequent invocation.

In theory this situation should not happen – in an ideal world, mappers should be stateless, to allow the MapReduce infrastructure to partition and re-order the mapper inputs without changing the result of the computation. However, in practice programmers do make use of state (such as frequency counters and temporary data structures) for efficiency or monitoring reasons, and typically via fields of the mapper class.

Consider for instance a mapper which increments a counter for each input row and produces output only on every n-th row. If we generate a filter that returns `true` for every n-th row and run the mapper on the filtered data set we will alarmingly have produced different output!

A simplistic solution to the problem would be to emit (trivial) filters that always return `true` for any `map` which depends on or modifies shared state. In practice, however, a surprising number of mappers access state and we would still like to generate non-trivial filters for these. Rhea does this by extending the definition of "output label" to include not only calls to the Hadoop API output methods but also instructions that could potentially affect shared state, such as method calls that involve mutable fields, field assignments and static methods, and also accesses of fields that are set in some part of the map method, and any methods of classes that could have some global observable effect, such as `java.lang.System` or Hadoop API methods. This ensures that the filter *approximates* the paths that could generate output in the mapper with a set of paths that (i) do not in any way depend on modifiable cross-invocation state; and (ii) do not contain any instructions that could themselves affect such shared state.

This simple approach is conservative but sound when there is use of state. More interestingly, this approach works well (i.e. generates non-trivial filters) with *common* uses of state. For example, in Listing 1, line 25 references the global field `geoLocationKey`. However, this happens in the same control flow block where the actual

```
1 public String select (LongWritable bcvar1,
2   Text bcvar2,
3   OutputCollector cvar3, Reporter bcvar4) {
4 String bcvar5 = bcvar2.toString();
5 String irvar0 = "\t";
6 StringTokenizer bcvar6
7     = StringTokenizer(bcvar5,irvar0);
8 int i = 0;
9 String filler = computeFiller(irvar0);
10 StringBuilder out = new StringBuilder();
11 String curr, aux;
12 while (bcvar6.hasMoreTokens()) {
13   curr = bcvar6.nextToken();
14   if (i == 2 || i == 1 || i == 0) {
15     aux = curr;
16   } else {
17     aux = filler;
18   };
19   if (bcvar6.hasMoreTokens()) {
20     out.append(aux).append(irvar0);
21   }
22   else {
23     out.append(aux);
24   }
25   i++;
26 }
27 return out.toString(); }
```

Listing 3: Column selector generated for GeoLocation

output instruction is located (line 28). Consequently, the generated filter is as precise as it could possibly be.

3.2 Column selection

So far we have described row filtering, where each input record is either suppressed entirely or passed unmodified to the computation. However, it is also valuable to suppress individual *columns* within rows. For example, in a top-K query, all rows must be examined to generate the output, but only a subset of the columns are relevant.

The Rhea filter generator analyzes the mapper function to produce a column selector method that transforms the input line into an output line with irrelevant column data suppressed. Column filtering may be combined with row filtering by using row filtering first and column selection on the remaining rows.

The static analysis for column selection is quite different from that used for row filtering. In Hadoop, mappers split each row (record) into columns (fields) in an application-specific manner. This is very flexible: it allows for different rows in the same file to have different numbers of columns. Mappers can also split the row into columns in different ways, e.g., using string splitting, or a tokenization library, or a regular expression matcher. This flexibility makes the problem of correctly removing irrelevant substrings challenging. Our approach is to detect and exploit common patterns of tokenization that we have encountered in many mappers. Our implementation supports tokenization based on Java's `StringTokenizer`

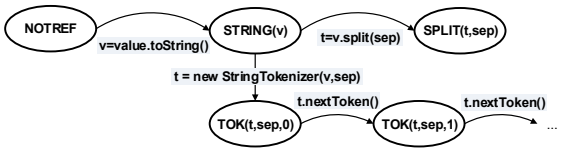


Figure 2: Transition system for column selector analysis

class and the `String.split()` API, but is easily extensible to other APIs.

For the GeoLocation map function in Listing 1, Rhea generates the column selector shown in Listing 3. The mapper only examines the first three tokens of the input (line 9–11 of Listing 1). The column selector captures this by retaining only the first three tokens. The output string is reassembled from the tokens after replacing all irrelevant tokens with a filler value, which is dynamically computed based on the separator used for tokenization.

Column filters always retain the token separators to ensure that the modified data is correctly parsed by the mapper. Dynamically computing the filler value allows us to deal with complex tokenization, e.g., using regular expressions. As a simple example, consider a comma-separated input line `"eve,usa,25"`. If the mapper splits the string at each comma, this can be transformed to `"eve,,25"`. However, if using a regular expression where multiple consecutive commas count as a single separator, `"eve,,25"` would be incorrect but `"eve,?,25"` would be correct. The `computeFiller` function correctly generates the filler according to the type of separator being used at run time.

The analysis assigns to each program point (label) in the mapper a state from a finite state machine which captures the current tokenization of the input. Figure 2 shows a simplified state machine that captures the use of the `StringTokenizer` class for tokenization. Essentially the input string can be in its initial state (`NOTREF`); it can be converted to a `String` (`STRING`); or this string can either have been split using `String.Split` (`SPLIT`) or converted to a `StringTokenizer` currently pointing to the *n*th token (`TOK(_,_,n)`).

The actual state machine used is slightly more complex. There is also an error state (not shown) that captures unexpected state transitions. The `TOK` state can also capture a non-deterministic state of the `StringTokenizer`: i.e., we can represent that at least *n* tokens have been extracted (but the exact upper bound is not known). The set of states is extended to form a lattice, which SAWJA's static analysis framework can use to map every program point to one of the states.

Assuming that no error states have been reached, we identify all program points that extract an input token that is then used elsewhere in the mapper. The tokenizer state at each of these points tells us which position(s) in the

input string this token could correspond to. The union of all these positions is the set of relevant token positions, i.e. columns. The filter generator then emits code for the column selector that tokenizes the input string, retains relevant columns, and replaces the rest with the filler.

Since our typical use cases involve unstructured data represented as `Text` values, we have focused on common string tokenization input patterns. Other use models do exist – for instance substring range selection – for which a different static analysis involving numerical constraints might be required [28]. Though entirely possible to design such analysis, we have focused on a few commonly used models. Our static analysis is able to detect when our parsing model is not directly applicable to the mapper implementation, in which case we conservatively accept the whole input and we are only in a position to get optimizations from row filtering.

Unlike row filtering, the presence of state in the mappers cannot compromise the soundness of the generated column filters, since column filters that conservatively retain *all dereferenced tokens* of the input, irrespectively of whether these tokens will be used in the control flow or to produce an output value, and whether different control flow paths assume different numbers of columns present in the input row.

3.3 Filter properties

Rhea's row and filter columns guarantee correctness in the sense that the output of the mapper is always the same for both filtered and unfiltered inputs. In addition we guarantee the following properties:

Filters are fully transparent Either row or column filtering can be done on a row-by-row basis, and filtered and unfiltered data can be interleaved arbitrarily. This allows filtering to be best-effort, i.e. it can be enabled/disabled on a fine-grained basis depending on available resources. It also allows filters to be chained, i.e. inserted anywhere in the data flow regardless of existing filters without compromising correctness.

Isolation and safety Filters cannot affect other code running in the same address space or the external environment. The generated filter code never includes I/O calls, system calls, dynamic class loads, or library invocations that affect global state outside the class containing the filter method.

Termination guarantees Column filters are guaranteed to terminate as they are produced only from a short column usage specification that we extract from the mapper using static analysis. Row filters may execute an arbitrary number of instructions and contain loops. Currently we dynamically disable row filters that consume excessive CPU resources. We could also statically guarantee termination by considering loops to be "output labels" that cause an early return of `true`, or use techniques to prove termination even in the presence of loops [8, 15].

As explained previously, our guarantees for column filters come with no assumptions whatsoever. Our row filter guarantees are with respect to our "prescribed" notion of state (system calls, mutable fields of the class, static fields, dynamic class loading). A mathematical proof of correctness would have to include a formalization of the semantics of JVM, and the MapReduce programming model. In this work we focus on the design and evaluation of our proposal and so we leave the formal verification as future work.

3.4 Applicability of static analysis

We collected the bytecode and source of 160 mappers from a variety of projects available on the internet to evaluate the applicability of our analysis. We ran these mappers through our tools and manually inspected the outputs to verify correctness. Approximately 50% of the mappers resulted in non-trivial row filters; the rest are always-true, due to the nature of the job or the use of state early on in the control flow. A common case is the use of state to measure and report the progress of input processing. In this case, we have to conservatively accept all input, even though reporting does not affect the output of the job. 26% of the mappers were amenable to our column tokenization models (the rest used the whole input, which often arises in libraries that operate on preprocessed data, or use a different parsing mechanism).

In our experiments the tasks of (i) identifying the mappers in the job, (ii) performing the static analysis on the mappers, (iii) generating filter source code, (iv) compiling the filter, and (v) generating the Rhea-aware Hadoop job, take a worst case time 4.8 seconds for a single mapper job on an Intel Xeon X5650 workstation. The static analysis part takes no more than 3 seconds.

In the next section we present the benefits of filtering for several jobs for which we had input data and were able to run more extensive experiments.

4 Evaluation

We ran two groups of experiments to evaluate the performance of Rhea. One group of experiments evaluates the performance within a single cloud data center, and the other aims to evaluate Rhea when using data stored in a remote data center.

4.1 Experimental setup

We ran the experiments, unless otherwise stated, on Windows Azure. A challenge for running the experiments within the data center is that we could not modify the Windows Azure storage to support the local execution of the filters we generated. To overcome this, for the experiments run in the single cloud scenarios, we used the filters generated to pre-filter the input data and then

stored it in Windows Azure storage. The bandwidth between the storage and the compute is the bottleneck resource, and this allows us to demonstrate the benefits of using Rhea. We micro-benchmark the filtering engine to demonstrate that it can sustain this throughput.

We use two metrics when measuring the performance of Rhea, selectivity and run time. Selectivity is the primary metric and captures how effective the Rhea filters are at reducing the data that needs to be transferred between the storage and compute. This is the primary metric of interest to a cloud provider, as this reduces the amount of data that is transferred across the core network between the storage clusters and compute. The second metric is run time, which is defined as the time to initialize and execute a Hadoop job on an existing cluster of compute VMs. Reducing run time is important in itself, but also because cloud computing VMs are charged per unit time, even if the VMs spend most of their time blocked on the network. Hence any reduction in execution time is important to the customer. The jobs that we run operated on a maximum input data set size of 100GB and all jobs ran in 15 minutes or less. Therefore, with per-hour VM charging, Rhea would provide little financial benefit when running a single job. However, if cloud providers move to finer grained pricing models or even per-job pricing models this will also have benefit; alternatively the customer could run more jobs within the same number of VM-hours and hence achieve cost savings per job. Unless otherwise stated, all graphs in this section show means of five identical runs, with error bars showing standard deviations.

To enable us to configure the experiments we measured the available storage-to-compute bandwidth for Windows Azure compute and scalable storage infrastructure, and for Amazon's infrastructure.

4.1.1 Storage-to-compute LAN bandwidth

The first set of experiments measured the storage-to-compute bandwidth for the Windows Azure data center by running a Hadoop MapReduce job with an empty mapper and no reducer. Running this shows the maximum rate at which input data can be ingested when there are no computational overheads at all. Each experiment read at least 60 GB to amortize any Hadoop start-up overheads. We also ran the experiment on Amazon's cloud infrastructure to see if there were significant differences in the storage-to-compute bandwidth across providers.

In the experiment we varied the number of instances used, between 4 and 16. We ran with extra large instances on both Amazon and Windows Azure, but also compared the performance with using small instances on Windows Azure. We found that bandwidth increases with the number of mappers per instance up to 16 mappers per instance, so we used 16 mappers per instance.

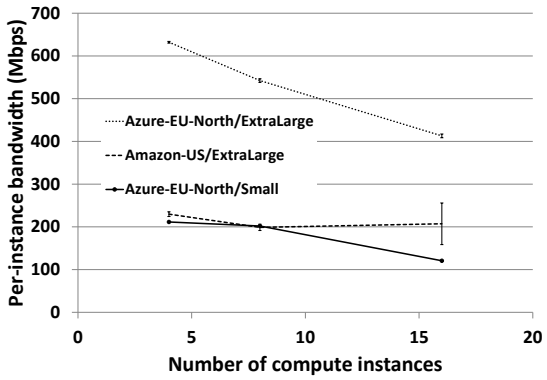


Figure 3: Storage-to-compute bandwidth in Windows Azure and Amazon cloud infrastructures. Labels show the provider, geographical location, and instance size used in each experiment.

Figure 3 shows the measured storage-to-compute transfer rate *per* compute instance. For Amazon the maximum per-instance ingress bandwidth is 230 Mbps, and the total is almost constant independent of the number of instances. For Windows Azure we observe that the peak ingress bandwidth is 631 Mbps when using 4 extra large instances. Contrary to the Amazon results, as the number of instances is increased the observed throughput per instance drops. Further, we observe that the small instance size on Windows Azure has significantly less bandwidth compared to the extra large instance.

Even in this best case (extra-large instances, no computational load, and a tuned number of mappers), the rate at which each compute instance can read data from storage is well below a single network adapter's bandwidth of 1 Gbps. More importantly it is lower than the rate at which most Hadoop computations can process data, making it the bottleneck. Hence, we would expect that reducing the amount of data transferred from storage to compute will not only provide network benefits but also, as we will show, run time performance improvements.

Based on these experiments, we run the experiments using 4 extra large compute instances on Azure-EU-North data center, each configured to run with 16 mappers per instance. This maximizes the bandwidth to the job, which is the worst case for Rhea. As the bandwidth becomes more constrained, through running on the Amazon infrastructure, by using smaller instances, or a larger number of instances the benefits of filtering will increase.

4.1.2 Job configuration

In all the experiments we use the 9 Hadoop jobs described in Section 2.3. Figure 4 shows the baseline results for input data size for each of the jobs and the run time when run in the Azure-EU-North data center with 4

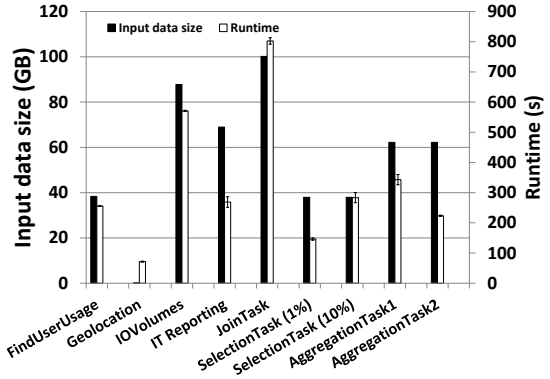


Figure 4: Input data sizes and job run times for the 9 example jobs when running on 4 extra large instances on Windows Azure without using Rhea.

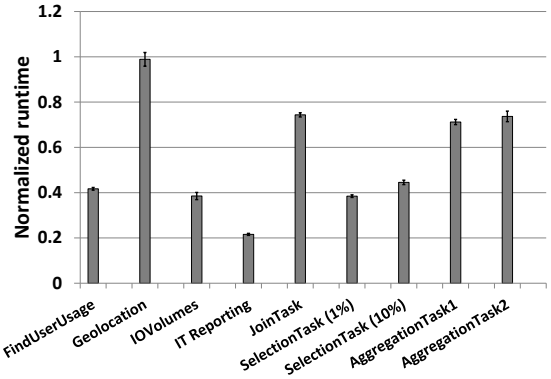


Figure 6: Job run time when using the Rhea filters normalized to the baseline execution time for the 9 example jobs when running on 4 extra large instances on Windows Azure.

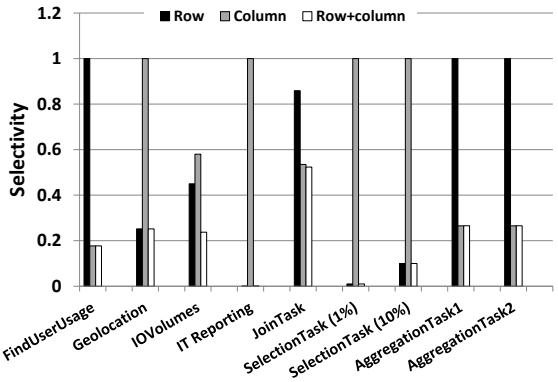


Figure 5: Selectivity for the row, column and combined filters for the 9 example jobs.

extra large compute instances without using Rhea. The input data size for the jobs varies from 90 MB–100 GB and the run times from 1–15 min. All but one job, GeoLocation, have an input data size of over 35 GB. To compare Rhea's effectiveness across this range of job sizes and run times, we show Rhea's data transfer sizes (Figure 5) and run times (Figure 6) normalized with respect to the results shown in Figure 4.

4.2 In cloud

The first set of experiments are run in a single cloud scenario: the data and compute are co-located in the same data center. The first results explore the selectivity of the filters produced by Rhea.

Selectivity For each of the nine jobs we take the input data and apply the row filter, the column filter, and the combined row and column filters and measure the selectivity. Selectivity is defined as the ratio of filtered data size to unfiltered data size; e.g. a selectivity of 1 means that no data reduction happened. Figure 5 shows the selectivity for row filters, for column filters, and the overall selectivity of row and column filters combined.

Figure 5 shows several interesting properties of Rhea. First, when using both row and column filtering across all jobs we observe a substantial reduction in the amount of input data transferred from the storage to the compute. In the worst case only 50% of the data was transferred. The majority of filters transferred only 25% of the data, and the most selective one only 0.005%, representing a reduction of 20,000 times the original data size. Therefore, in general the approach provides significant gains.

We also see that for five jobs the column selectivity is 1.0. In these cases no column filter was generated by Rhea. In three cases, the row selectivity is 1.0. In these cases, row filters were generated but did not suppress any rows. On examination, we found that the filters were essentially a check for a validly formed line of input (a common check in many mappers). Since our test inputs happened to consist only of valid lines, none of the lines were suppressed at run time. Note that a filter with poor selectivity can easily be disabled at run time without modifying or even restarting the computation.

Runtime Next we look at the impact of filtering on the execution time of jobs running in Windows Azure.

Figure 6 shows the run time for the nine jobs when using the Rhea filters normalized to the time taken to the baseline. For half the jobs we observe a speed up of over a factor of two. For four of the remaining jobs we observe that the time taken is 75% or lower compared to the baseline. The outlier in the GeoLocation example which, despite the data selectivity being high, has an identical run time. This is because the data set is small and the run time setup overheads dominate.

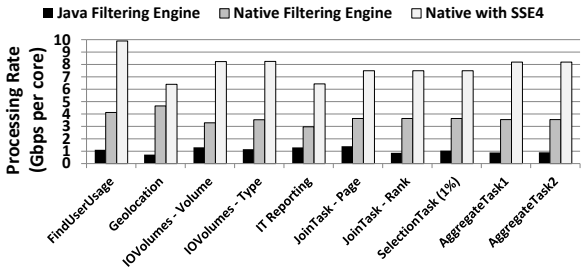


Figure 7: Input data rates achieved by filtering for row and column filters in Java and declarative column filters alone in two native filtering engines. Observe that two of the jobs contain two mappers each, for which we measure filtering performance independently.

Filtering engine These experiments run with pre-filtered data as we can not modify the Windows Azure storage layer. Separately, we micro-benchmarked the throughput of the filtering engine. Our goal is to understand if filtering can become a bottleneck for the job and hence slow down job run times. Although filtering still reduces overall network bandwidth usage, we would disable such filters to prevent individual jobs from slowing down.

Consider a modest storage server with 2 cores and a 1 Gbps network adapter. Assuming a server transmitting at full line rate, the filters should process data at an input rate of 1 Gbps or higher, to guarantee that filtering will not degrade job performance. In practice, with a large number of compute and storage servers, core network bandwidth is the bottleneck and the server is unlikely to achieve full line rate. The black bars in Figure 7 show the filtering throughput per core measured in isolation, with both input and output data stored in memory, and both row and column filters enabled for all jobs. All the filters run faster than 500 Mbps per core (on an Intel Xeon X5650 processor), showing that even with conservative assumptions filtering will not degrade job performance.

We have also experimented with *declarative* rather than executable column filters, which allows us to use a fast native filtering engine (no JVM). Recall that the static analysis for column filtering generates a description of the tokenization process (e.g. separator character, regular expression) and a list of, e.g., integers that identify the columns that are dereferenced by the mapper. Instead of converting this to Java code, we encode it as a symbolic filter which is interpreted by a fast generic engine written in C. This engine is capable of processing inputs 2.5-9x faster than the Java filtering engine (median 3.7x) (Figure 7). We have further optimized the C engine using the SSE4 instruction set. The performance increased to 5-17x faster than the Java filtering engine (median 8.6). In addition to performance, the native engine is small and self-contained, and easily isolated, but

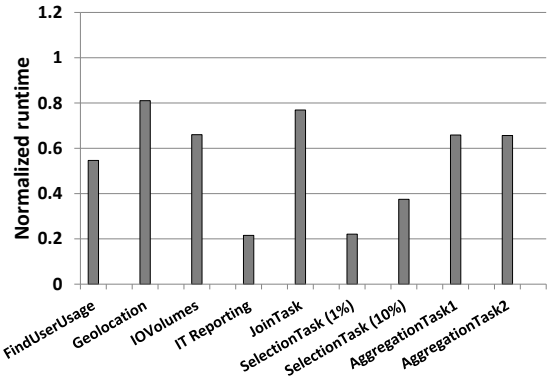


Figure 8: Job run times when using the Rhea filters normalized to the baseline execution time for the 9 example jobs fetching data across the WAN

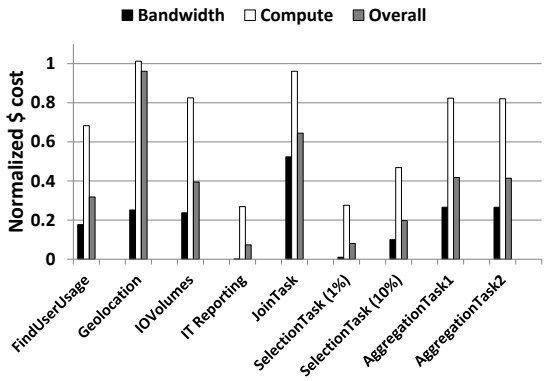


Figure 9: Dollar costs when using the Rhea filters normalized to the baseline cost for the 9 example jobs fetching data across the WAN

it does not perform row filtering. For row filtering currently we still use the slower Java filtering engine: row filters can perform arbitrary computations and we currently have no mechanism for converting them from Java to a declarative representation.

The performance numbers reported in Figure 7 are per processor core. It is straightforward to run in parallel multiple instances of the same filter, or even different filters. The system performance of filtering increases linearly with the number of cores, assuming of course enough I/O capacity for reading input data and network capacity for transmitting filtered data.

4.3 Cross cloud with online filtering

There are several scenarios where data must be read across the WAN. Data could be stored on-premise and occasionally accessed by a computation run on an elastic cloud infrastructure for cost or scalability reasons.

Alternatively, data could be in cloud storage but computation run on-premise: for computations that do not need elastic scalability and with a heavy duty cycle, on-premises computation is cheaper than renting cloud VMs. A third scenario is when the data are split across multiple providers or data centers. For example, a job might join a public data set available on one provider with a private data set stored on a different provider. The computation must run on one of the providers and access the data on the other one over the WAN.

WAN bandwidth is even scarcer than LAN bandwidth, and using cloud storage incurs egress charges. Thus using Rhea reduces both LAN and WAN traffic if the data are split across data centers. Since, we have already evaluated the LAN scenario, we will now evaluate the effects of filtering WAN traffic with Rhea in isolation. To do this we run the same nine jobs with the only difference being that the computations are run in the Azure-US-West data center and the storage is in the Azure-EU-North data center[2]. Rhea filters are deployed in a single large compute instance running as a filtering proxy in the Azure-EU-North data center's compute cluster.

Run time Figure 8 shows the run time when Rhea filtering is used, normalized to the baseline run time with no filtering. In general the results are similar to the LAN case. In all cases the CPU utilization reported on the filtering proxy was low (under 20% always). Thus the proxy is never the bottleneck. In most cases the WAN bandwidth is the bottleneck and the reduction in run time is due to the filter selectivity. However, for very selective filters (*IT Reporting*), the bottleneck is the data transfer from the storage layer to the filtering proxy over the LAN rather than the transfer from the proxy to the WAN. In this case the run time reduction reflects the ratio of the WAN egress bandwidth, to the LAN storage-to-compute bandwidth achieved by the filtering proxy.

Dollar costs In the WAN case, dollar costs reduce both for compute instances and also for egress bandwidth. While Rhea uses more compute instances (by adding a filtering instance) it significantly reduces egress bandwidth usage. Figure 9 shows the bandwidth, compute, and overall dollar costs of Rhea, each normalized to the corresponding value when not using Rhea. We use the standard Windows Azure charges of US$0.96 per hour for an extra-large instance and US$0.12 per GB of egress bandwidth. Surprisingly the compute costs also go down when using Rhea, even though it uses 5 instances per job rather than 4. This is because overall run times are reduced (again assuming per-second rather than per-hour billing, since most of our jobs take well under an hour to run). Thus compute costs are reduced in line with run time reductions and egress bandwidth charges in line with data reduction. In general, we expect the effect of egress bandwidth to dominate since computation is cheap relative to egress bandwidth: one hour of compute costs the same as only 8 GB of data egress. Of course, if filtering were offered at the storage servers then it would simply use spare computing cycles there and there would be no need to pay for a filtering VM instance.

[2]The input data sets for FindUserUsage and ComputeIOVolumes are too large to run in a reasonable time in this configuration. Hence for these two jobs we use a subset of the data, i.e. 1 hour's event logs rather than 1 day's.

5 Related work

There is a large body of work optimizing the performance of MapReduce, by better scheduling of jobs [21] and by handling of stragglers and failures [2, 40]. We are orthogonal to this work, aiming to minimize bandwidth between storage and compute.

Pyxis is a system for automatically partitioning database applications [7]. It uses profiling to identify opportunities for splitting a database application between server and application nodes, so that data transfers are minimized (like Rhea), but also *control transfers* are minimized. Unlike Rhea, Pyxis uses *state-aware* program partitioning. The evaluation has been done of Java applications running against MySQL. Compared to Rhea, the concerns are different: database applications might be more interactive (with more control transfers) than MapReduce data analytics programs; moreover in our setting we consider partitioning to be just an optimization that can opportunistically be enabled or disabled on the storage, even during the execution of a job and hence we do not modify the original job and make sure that the extracted filters are stateless. On the other hand, the optimization problem that determines the partitioning can take into account the available CPU budget at the database nodes, a desirable feature for Rhea as well.

MANIMAL is an analyzer for MapReduce jobs [25]. It uses static analysis techniques similar to Rhea's to generate an "index-generation program" which is run *off-line* to produce an indexed and column-projected version of the data. Index-generation programs must be run to completion on the entire data set to show any benefit, and must be re-run whenever additional data is appended. The entire data set must be read by Hadoop compute nodes and then the index written back to storage. This is not suitable for our scenario where there is limited bandwidth between storage and compute. By contrast, Rhea filters are *on-line* and have no additional overheads when fresh data are appended. Furthermore, MANIMAL uses logical formulas to encode the "execution descriptors" that perform row filtering by selecting appropriately indexed versions of the input data. Rhea filters can encode arbitrary Boolean functions over input rows.

Hadoop2SQL [22] allows the efficient execution of Hadoop code on a SQL database. The high-level goal

is to transform a Hadoop program into a SQL query or, if the entire program cannot be transformed, parts of the program. This is achieved by using static analysis. The underlying assumption is that by pushing the Hadoop query into the SQL database it will be more efficient. In contrast, the goal of Rhea is to still enable Hadoop programs to run on a cluster against any store that can currently be used with Hadoop.

Using static analysis techniques to unravel properties of user-defined functions and exploit opportunities for optimizations is an area of active research. In the SUDO system [42], a simple static analysis of user-defined functions determines whether they preserve the input data partition properties. This information is used to optimize the shuffling stage of a distributed SCOPE job. In the context of the Stratosphere project [19], code analysis determines algebraic properties of user-defined functions and an optimizer exploits them to rewrite and further optimize the query operator graph. The NEMO system [16] also treats UDFs as open-boxes and tries to identify opportunities for applying more traditional "whole-program" optimizations, such as function and type specialization, code motion, and more. This could potentially be used to "split" mappers rather than "extract" filters, i.e. modify the mapper to avoid repeating the computation of the filter. However this is very difficult to do automatically, and indeed with NEMO manual modification is required to create such a split. Further, it means that filters can no longer be dynamically and transparently disabled since they are now an indispensable part of the application.

In the storage field the closest work is on Active Disks [20, 32]. Here compute resources are provided directly in the hard disk and a program is partitioned to run on the server and on the disks. A programmer is expected to manually partition the program, and the operations performed on the disk transform the data read from it. Rhea pushes computation into the storage layer but it does not require any explicit input from the programmer.

Inferring the schema of unstructured or semi-structure data is an interesting problem, especially for mining web pages [9, 10, 27]. Due to the difficulty of constructing hand-coded wrappers, previous work focused on automated ways to create those wrappers, often with the use of examples [27]. In Rhea, the equivalent hand-coded wrappers are actually embedded in the code of the mappers, and our challenge is to extract them in order to generate the filters. Moreover, Rhea deals with very flexible schemas (e.g. different rows may have different structure); our goal is not to interpret the data, but to extract enough information to construct the filters.

Rhea reduces the amount of data transferred by filtering the input data. Another approach to reduce the bytes transferred is with compression [33, 35]. We have found that compression complements filtering to further reduce the amount of bytes transferred in our data sets. Compression though requires changes to the user code, and increases the processing overhead at the storage nodes.

Regarding the static analysis part of this work, there is a huge volume of work on dependency analysis for slicing from the early 80's [36], to elaborate inter-procedural slicing [17]. More recently, Wiedermann *et al.* [37, 38] studied the program of extracting queries from imperative programs that work on *structured* data that adhere to a database schema. The techniques used are similar as ours here – an abstract interpretation framework keeps track of the used structure and the paths of the imperative program that perform output or update the state. A key difference is that Rhea targets unstructured text inputs, so a separate analysis is required to identify the parts of the input string that are used in a program. Moreover our tool extracts programs in a language as expressive as the original mapper – as opposed to a specialized query language. This allows us to be very flexible in the amount of computation that we can embed into the filter and push close to the data.

6 Conclusions

We have described Rhea, a system that automatically generates executable storage-side filters for unstructured data processing in the cloud. The filters encode the implicit data selectivity, in terms of row and column, for map functions in Hadoop jobs. They are created by performing static analysis on Java byte code.

We have demonstrated that Rhea filtering yields significant savings in the data transferred between storage and compute for a variety of realistic Hadoop jobs. Reduced bandwidth usage leads to faster job run times and lower dollar costs when data is transferred cross-cloud. The filters have several desirable properties: they are transparent, safe, lightweight, and best-effort. They are guaranteed to have no false negatives: all data used by a map job will be passed through the filter. Filtering is strictly an optimization. At any point in time the filter can be stopped and the remaining data returned unfiltered transparently to Hadoop.

We are currently working on generalizing Rhea to support other format such as binary formats, XML, and compressed text, as well as data processing tools and runtimes other than Hadoop and Java.

Acknowledgments

We thank the reviewers, and our shepherd Wenke Lee, who provided valuable feedback and advice. Thanks to the Microsoft Hadoop product team for valuable discussions and resources, and in particular Tim Mallalieu, Mike Flasko, Steve Maine, Alexander Stojanovic, and Dave Vronay.

References

[1] *Amazon Simple Storage Service (Amazon S3)*. `http://aws.amazon.com/s3/`. Accessed: 08/09/2011.

[2] G. Ananthanarayanan et al. "Reining in the Outliers in Map-Reduce Clusters using Mantri". *Operating Systems Design and Implementation (OSDI)*. USENIX, 2010.

[3] *Apache Cassandra*. `http://cassandra.apache.org/`. Accessed: 03/10/2011.

[4] B. Calder et al. "Windows Azure Storage: a highly available cloud storage service with strong consistency". *Proc. of 23rd Symp. on Operating Systems Principles (SOSP)*. ACM, 2011.

[5] R. Chaiken et al. "SCOPE: Easy and Effcient Parallel Processing of Massive Datasets". *VLDB*. 2008.

[6] Y. Chen et al. "The Case for Evaluating MapReduce Performance Using Workload Suites". *MASCOTS*. IEEE Computer Society, 2011.

[7] A. Cheung et al. "Automatic partitioning of database applications". *Proc. VLDB Endow.* 5.11 (2012).

[8] B. Cook, A. Podelski, and A. Rybalchenko. "Termination proofs for systems code". *Proc. of the SIGPLAN conf. on Programming Language Design and Implementation (PLDI)*. ACM, 2006.

[9] V. Crescenzi and G. Mecca. "Automatic Information Extraction from Large Websites". *J. ACM* 51.5 (2004).

[10] V. Crescenzi, G. Mecca, and P. Merialdo. "RoadRunner: Towards Automatic Data Extraction from Large Web Sites". *Proc. of 27th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann Publishers Inc., 2001.

[11] T. von Eicken. *Network performance within Amazon EC2 and to Amazon S3*. `http://blog.rightscale.com/2007/10/28/network-performance-within-amazon-ec2-and-to-amazon-s3/`. Accessed: 08/09/2011. 2007.

[12] S. L. Garfinkel. *An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS*. Tech. rep. Harvard University, 2007.

[13] A. F. Gates et al. "Building a high-level dataflow system on top of Map-Reduce: the Pig experience". *Proc. VLDB Endow.* 2.2 (2009).

[14] A. G. Greenberg et al. "VL2: a scalable and flexible data center network". *SIGCOMM*. Ed. by P. Rodriguez et al. ACM, 2009.

[15] S. Gulwani, K. K. Mehra, and T. Chilimbi. "SPEED: precise and efficient static estimation of program computational complexity". *Proc. of 36th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*. ACM, 2009.

[16] Z. Guo et al. "Nemo: Whole Program Optimization for Distributed Data-Parallel Computation." *Proc. of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2012.

[17] S. Horwitz, T. Reps, and D. Binkley. "Interprocedural slicing using dependence graphs". *SIGPLAN Not.* 39 (4 2004).

[18] L. Hubert et al. "Sawja: Static Analysis Workshop for Java". *Formal Verification of Object-Oriented Software*. Ed. by B. Beckert and C. Marché. Springer Berlin / Heidelberg, 2011.

[19] F. Hueske et al. "Opening the black boxes in data flow optimization". *Proc. VLDB Endow.* 5.11 (2012).

[20] L. Huston et al. "Diamond: A Storage Architecture for Early Discard in Interactive Search". *FAST*. USENIX, 2004.

[21] M. Isard et al. "Quincy: Fair Scheduling for Distributed Computing Clusters". *Proc. of 22nd ACM Symposium on Operating Systems Principles (SOSP)*. 2009.

[22] M.-Y. Iu and W. Zwaenepoel. "HadoopToSQL: A MapReduce query optimizer". *EuroSys'10*. 2010.

[23] S. Iyer. *Geo Location Data From DB-Pedia*. `http://downloads.dbpedia.org/3.3/en/geo_en.csv.bz2`. Accessed: 22/09/2011.

[24] S. Iyer. *Map Reduce Program to group articles in Wikipedia by their GEO location*. `http://code.google.com/p/hadoop-map-reduce-examples/wiki/Wikipedia_GeoLocation`. Accessed: 08/09/2011. 2009.

[25] E. Jahani, M. J. Cafarella, and C. Ré. "Automatic Optimization for MapReduce Programs". *PVLDB* 4.6 (2011).

[26] R. Jhala and R. Majumdar. "Path slicing". *Proc. of the 2005 ACM SIGPLAN conf. on Programming Language Design and Implementation (PLDI)*. ACM, 2005.

[27] N. Kushmerick, D. S. Weld, and R. B. Doorenbos. "Wrapper Induction for Information Extraction". *IJCAI (1)*. Morgan Kaufmann, 1997.

[28] A. Miné. "The octagon abstract domain". *Higher Order Symbol. Comput.* 19.1 (2006).

[29] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[30] A. Pavlo et al. "A comparison of approaches to large-scale data analysis". *Proc. of 35th SIGMOD intl conf on Management of data*. ACM, 2009.

[31] *Public Data Sets on AWS*. `http://aws.amazon.com/publicdatasets/`. Accessed: 03/05/2012.

[32] E. Riedel et al. "Active Disks for Large-Scale Data Processing". *Computer* 34 (6 2001).

[33] *snappy: A fast compressor/decompressor*. `http://code.google.com/p/snappy/`. Accessed: 03/05/2012.

[34] A. Thusoo et al. "Hive - a petabyte scale data warehouse using Hadoop". *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*. IEEE, 2010.

[35] B. D. Vo and G. S. Manku. "RadixZip: linear time compression of token streams". *Proc of 33rd intl. conf. on Very Large Data Bases (VLDB)*. VLDB Endowment, 2007.

[36] M. Weiser. "Program slicing". *Proc. of 5th intl. conf. on Software Engineering (ICSE)*. IEEE Press, 1981.

[37] B. Wiedermann and W. R. Cook. "Extracting queries by static analysis of transparent persistence". *Proc. of 34th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*. ACM, 2007.

[38] B. Wiedermann, A. Ibrahim, and W. R. Cook. "Interprocedural query extraction for transparent persistence". *Proc. of 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA)*. ACM, 2008.

[39] *Windows Azure Storage*. `http://www.microsoft.com/windowsazure/features/storage/`. Accessed: 08/09/2011.

[40] M. Zaharia et al. "Improving MapReduce Performance in Heterogeneous Environments". *Operating Systems Design and Implementation (OSDI)*. USENIX, 2008.

[41] *Zebra Reference Guide*. `http://pig.apache.org/docs/r0.7.0/zebra_reference.html`. Accessed: 22/09/2011. 2011.

[42] J. Zhang et al. "Optimizing data shuffling in data-parallel computation by understanding user-defined functions". *Proc. of 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.

Robustness in the Salus scalable block store

Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam,
Lorenzo Alvisi, and Mike Dahlin
The University of Texas at Austin

Abstract: This paper describes Salus, a block store that seeks to maximize simultaneously both scalability and robustness. Salus provides strong end-to-end correctness guarantees for read operations, strict ordering guarantees for write operations, and strong durability and availability guarantees despite a wide range of server failures (including memory corruptions, disk corruptions, firmware bugs, etc.). Such increased protection does not come at the cost of scalability or performance: indeed, Salus often actually outperforms HBase (the codebase from which Salus descends). For example, Salus' active replication allows it to halve network bandwidth while increasing aggregate write throughput by a factor of 1.74 compared to HBase in a well-provisioned system.

1 Introduction

The primary directive of storage—not to lose data—is hard to carry out: disks and storage sub-systems can fail in unpredictable ways [7, 8, 18, 23, 34, 37], and so can the CPUs and memories of the nodes that are responsible for accessing the data [33, 38]. Concerns about robustness become even more pressing in cloud storage systems, which appear to their clients as black boxes even as their larger size and complexity create greater opportunities for error and corruption.

This paper describes the design and implementation of Salus,[1] a scalable block store in the spirit of Amazon's Elastic Block Store (EBS) [1]: a user can request storage space from the service provider, mount it like a local disk, and run applications upon it, while the service provider replicates data for durability and availability.

What makes Salus unique is its dual focus on *scalability* and *robustness*. Some recent systems have provided end-to-end correctness guarantees on distributed storage despite arbitrary node failures [13, 16, 31], but these systems are not scalable—they require each correct node to process at least a majority of updates. Conversely, scalable distributed storage systems [3, 4, 6, 11, 14, 20, 25, 30, 43] typically protect some subsystems like disk storage with redundant data and checksums, but fail to protect the entire path from client PUT to client GET, leaving them vulnerable to single points of failure that can cause data corruption or loss.

Salus provides strong end-to-end correctness guarantees for read operations, strict ordering guarantees for write operations, and strong durability and availability guarantees despite a wide range of server failures (including memory corruptions, disk corruptions, firmware bugs, etc), and leverages an architecture similar to scalable key-value stores like Bigtable [14] and HBase [6] towards scaling these guarantees to thousands of machines and tens of thousands of disks.

Achieving this unprecedented combination of robustness and scalability presents several challenges.

First, to build a high-performance block store from low-performance disks, Salus must be able to write different sets of updates to multiple disks in parallel. Parallelism, however, can threaten the basic consistency requirement of a block store, as "later" writes may survive a crash, while "earlier" ones are lost.

Second, aiming for efficiency and high availability at low cost can have unintended consequences on robustness by introducing single points of failure. For example, in order to maximize throughput and availability for reads while minimizing latency and cost, scalable storage systems execute read requests at just one replica. If that replica experiences a *commission failure* that causes it to generate erroneous state or output, the data returned to the client could be incorrect. Similarly, to reduce cost and for ease of design, many systems that replicate their storage layer for fault tolerance (such as HBase) leave unreplicated the computation nodes that can modify the state of that layer: hence, a memory error or an errant PUT at a single HBase region server can irrevocably and undetectably corrupt data (see §5.1).

Third, additional robustness should ideally not result in higher replication cost. For example, in a perfect world Salus' ability to tolerate commission failures would not require any more data replication than a scalable key-value store such as HBase already employs to ensure durability despite omission failures.

To address these challenges Salus introduces three novel ideas: pipelined commit, active storage, and scalable end-to-end verification.

Pipelined commit. Salus' new pipelined commit protocol allows writes to proceed in parallel at multiple disks but, by tracking the necessary dependency information during failure-free execution, guarantees that, despite failures, the system will be left in a state consistent with the ordering of writes specified by the client.

Active storage. To prevent a single computation node from corrupting data, Salus replicates both the storage and the computation layer. Salus applies an update to the system's persistent state only if the update is agreed up-

[1] Salus is the Roman goddess of safety and welfare

on by *all* of the replicated computation nodes. We make two observations about active storage. First, perhaps surprisingly, replicating the computation nodes can actually improve system performance by moving the computation near the data (rather than vice versa), a good choice when network bandwidth is a more limited resource than CPU cycles. Second, by requiring the *unanimous consent* of all replicas before an update is applied, Salus comes near to its perfect world with respect to overhead: Salus remains safe (i.e. keeps its blocks consistent and durable) despite two *commission* failures with just three-way replication—the same degree of data replication needed by HBase to tolerate two permanent *omission* failures. The flip side, of course, is that insisting on unanimous consent can reduce the times during which Salus is live (i.e. its blocks are available)—but liveness is easily restored by replacing the faulty set of computation nodes with a new set that can use the storage layer to recover the state required to resume processing requests.

Scalable end-to-end verification. Salus maintains a Merkle tree [32] for each volume so that a client can validate that each GET request returns consistent and correct data: if not, the client can reissue the request to another replica. Reads can then safely proceed at a single replica without leaving clients vulnerable to reading corrupted data; more generally, such end-to-end assurances protect Salus clients from the opportunities for error and corruption that can arise in complex, black-box cloud storage solutions. Further, Salus' Merkle tree, unlike those used in other systems that support end-to-end verification [19, 26, 31, 41], is scalable: each server only needs to keep the sub-tree corresponding to its own data, and the client can rebuild and check the integrity of the whole tree even after failing and restarting from an empty state.

We have prototyped Salus by modifying the HBase key-value store. The evaluation confirms that Salus can tolerate servers experiencing commission failures like memory corruption, disk corruption, etc. Although one might fear the performance price to be paid for Salus' robustness, Salus' overheads are low in all of our experiments. In fact, despite its strong guarantees, Salus often *outperforms* HBase, especially when disk bandwidth is plentiful compared to network bandwidth. For example, Salus' active replication allows it to halve network bandwidth while increasing aggregate write throughput by a factor of 1.74 in a well-provisioned system.

2 Requirements and model

Salus provides the abstraction of a large collection of virtual disks, each of which is an array of fixed-sized blocks. Each virtual disk is a *volume* that can be mounted by a client running in the datacenter that hosts the volume. The volume's size (e.g., several hundred GB to several hundred TB) and block size (e.g., 4 KB to 256 KB) are specified at creation time

A volume's interface supports GET and PUT, which on a disk correspond to read and write. A client may have many such commands outstanding to maximize throughput. At any given time, only one client may mount a volume for writing, and during that time no other client can mount the volume for reading. Different clients may mount and write different volumes at the same time, and multiple clients may simultaneously mount a read-only snapshot of a volume.

We explicitly designed Salus to support only a single writer per volume for two reasons. First, as demonstrated by the success of Amazon EBS, this model is sufficient to support disk-like storage. Second, we are not aware of a design that would allow Salus to support multiple writers while achieving its other goals: strong consistency,[2] scalability, and end-to-end verification for read requests.

Even though each volume has only a single writer at a time, a distributed block store has several advantages over a local one. Spreading a volume across multiple machines not only allows disk throughput and storage capacity to exceed the capabilities of a single machine, but balances load and increases resource utilization.

To minimize cost, a typical server in existing storage deployments is relatively storage heavy, with a total capacity of up to 24 TB [5, 42]. We expect a storage server in a Salus deployment to have ten or more SATA disks and two 1 Gbit/s network connections. In this configuration disk bandwidth is several times more plentiful than network bandwidth, so the Salus design seeks to minimize network bandwidth consumption.

2.1 Failure model

Salus is designed to operate on an unreliable network with unreliable nodes. The network can drop, reorder, modify, or arbitrarily delay messages.

For storage nodes, we assume that 1) servers can crash and recover, temporarily making their disks' data unavailable (transient omission failure); 2) servers and disks can fail, permanently losing all their data (permanent omission failure); 3) disks and the software that controls them can cause corruption, where some blocks are lost or modified, possibly silently [35] and servers can experience memory corruption, software bugs, etc, sending corrupted messages to other nodes (commission failure). When calculating failure thresholds, we only take into account commission failures and permanent omission failures. Transient omission failures are not treated as failures: in asynchronous systems a node that fails and recovers is indistinguishable from a slow node.

In line with Salus' aim to provide end-to-end robustness guarantees, we do not try to explicitly enumerate and patch all the different ways in which servers can fail. Instead, we design Salus to tolerate arbitrary fail-

[2] More precisely, *ordered commit* (defined in §2.2) which for multiple clients implies FIFO-compliant linearizability.

ures, both of omission, where a faulty node fails to perform actions specified by the protocol, such as sending, receiving or processing a message; and of commission [16], where a faulty node performs arbitrary actions not called for by the protocol. However, while we assume that faulty nodes will potentially generate arbitrarily erroneous state and output, given the data center environment we target we explicitly do not attempt to tolerate cases where a malicious adversary controls some of the servers. Hence, we replace the traditional BFT assumption that *faulty nodes cannot break cryptographic primitives* [36] with the stronger (but fundamentally similar) assumption that *a faulty node never produces a checksum that appears to be a correct checksum produced by a different node.* In practice, this means that where in a traditional Byzantine-tolerant system [12] we might have used signatures or arrays of message authentication codes (MACs) with pairwise secret keys, we instead *weakly sign* communication using checksums salted with the checksum creator's well-known ID.

Salus relies on weak synchrony assumptions for both safety and liveness. For safety, Salus assumes that clocks are sufficiently synchronized that a ZooKeeper lease is never considered valid by a client when the server considers it invalid. Salus only guarantees liveness during *synchronous intervals* where messages sent between correct nodes are received and processed within some timeout [10].

2.2 Consistency model

To be usable as a virtual disk, Salus tries to preserve the *standard disk semantics* provided by physical disks. These semantics allow some requests to be marked as *barriers*. A disk must guarantee that all requests received before a barrier are committed before the barrier, and all requests received after the barrier are committed after the barrier. Additionally, a disk guarantees *freshness*: a read to a block returns the latest committed write to that block.

During normal operation (up to two commission or omission failures), Salus guarantees both freshness and a property we call *ordered-commit*: by the time a request *R* is committed, all requests that were received before *R* have committed. Note that ordered-commit eliminates the need for explicit barriers since every write request functions as a barrier. Although we did not set out to achieve ordered-commit and its stronger guarantees, Salus provides them without any noticeable effect on performance.

Under severe failures Salus provides the weaker *prefix semantics*: in these circumstances, a client that crashes and restarts may observe only a prefix of the committed writes; a tail of committed writes may be lost. This semantics is not new to Salus: it is the semantics familiar to every client that interacts with a crash-prone server that acknowledges writes immediately but logs them asynchronously; it is also the semantics to which every other geo-replicated storage systems we know of [11, 29, 31] retreats when failures put it under duresse. The reason is simple: while losing writes is always disappointing, prefix semantics has at least the merit of leaving the disk in a legal state. Still, data loss should be rare, and Salus falls back on prefix semantics only in the following scenario: the client crashes, one or more of the servers suffer at the same time a commission failure, and the rest of the servers are unavailable. If the client does not fail or at least one server is correct and available, Salus continues to guarantee standard disk semantics.

Salus mainly focuses on tolerating arbitrary failures of server-side storage systems, since they entail most of the complexity and are primarily responsible for preserving the durability and availability of data. Client commission failures can also be handled using replication, but this falls beyond the scope of this paper.

3 Background

Salus' starting point is the scalable architecture of HBase/HDFS, which Salus carefully modifies to boost robustness without introducing new bottlenecks. We chose the HBase/HDFS architecture for three main reasons: first, because it provides a key-value interface that can be easily modified to support a block store; second, because it has a large user base that includes companies such as Yahoo!, Facebook, and Twitter; and third because, unlike other successful large-scale storage systems with similar architectural features, such as Windows Azure [11] and Google's Bigtable/GFS [14, 20], HBase/HDFS is open source.

HDFS HDFS [39] is an append-only distributed file system. It stores the system metadata in a *NameNode* and replicates the data over a set of *datanodes*. Each file consists of a set of blocks and HDFS ensures that each block is replicated across a specified number of datanodes (three by default) despite datanode failures. HDFS is widely used, primarily because of its scalability.

HBase HBase [6] is a distributed key-value store. It exports the abstraction of tables accessible through a PUT/GET interface. Each table is split into multiple *regions* of non-overlapping key-ranges (for load balancing). Each region is assigned to one *region server* that is responsible for all requests to that region. Region servers use HDFS as a storage layer to ensure that data is replicated persistently across enough nodes. Additionally, HBase uses a *Master* node to manage the assignment of key-ranges to various region servers.

Region servers receive clients' PUT and GET requests and transform them into equivalent requests that are appropriate for the append-only interface exposed by HDFS. On receiving a PUT, a region server logs the request to a write-ahead-log stored on HDFS and updates its

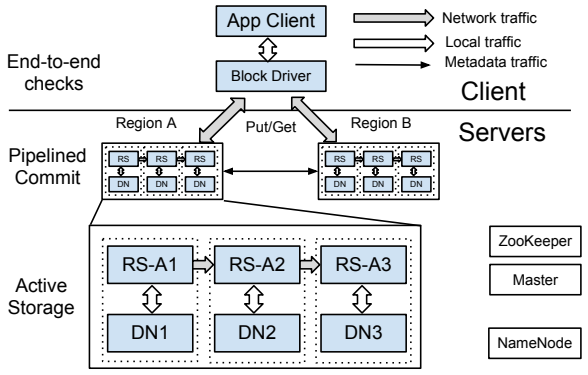


Fig. 1: The architecture of Salus. Salus differs from HBase in three key ways. First, Salus' block driver performs end-to-end checks to validate the GET reply. Second, Salus performs pipelined commit across different regions to ensure ordered commit. Third, Salus replicates region servers via active storage to eliminate spurious state updates. For efficiency, Salus tries to co-locate the replicated region servers with the replicated datanodes (DNs).

sorted, in-memory map (called *memstore*) with the new PUT. When the size of the memstore exceeds a predefined threshold, the region server *flushes* the memstore to a *checkpoint* file stored on HDFS.

On receiving a GET request for a key, the region server looks up the key in its memstore. If a match is found, the region server returns the corresponding value; otherwise, it looks up the key in various checkpoints, starting from the most recent one, and returns the first matching value. Periodically, to minimize the storage overheads and the GET latency, the region server performs *compaction* by reading a number of contiguous checkpoints and merging them into a single checkpoint.

ZooKeeper ZooKeeper [22] is a replicated coordination service. It is used by HBase to ensure that each key-range is assigned to at most one region server.

4 The design of Salus

The architecture of Salus, as Figure 1 shows, bears considerable resemblance to that of HBase. Like HBase, Salus uses HDFS as its reliable and scalable storage layer, partitions key ranges within a table in distinct regions for load balancing, and supports the abstraction of a region server responsible for handling requests for the keys within a region. As in HBase, blocks are mapped to their region server through a Master node, leases are managed using ZooKeeper, and Salus clients need to install a *block driver* to access the storage system, not unlike the client library used for the same purpose in HBase. These similarities are intentional: they aim to retain in Salus the ability to scale to thousands of nodes and tens of thousands of disks that has secured HBase's success. Indeed, one of the main challenges in designing Salus was to achieve its robustness goals (strict ordering guarantees for write operations across multiple disks, end-to-end correctness guarantees for read operations, strong availability and durability guarantees despite arbitrary failures) without perturbing the scalability of the original HBase design. With this in mind, we have designed Salus so that, whenever possible, it buttresses architectural features it inherits from HBase—and does so scalably. So, the core of Salus' active storage is a three-way replicated region server (RRS), which upgrades the original HBase region server abstraction to guarantee safety despite up to two arbitrary server failures. Similarly, Salus' end-to-end verification is performed within the familiar architectural feature of the block driver, though upgraded to support Salus' scalable verification mechanisms.

Figure 1 also helps describe the role played by our novel techniques (pipelined commit, scalable end-to-end verification, and active storage) in the operation of Salus.

Every client request in Salus is mediated by the block driver, which exports a virtual disk interface by converting the application's API calls into Salus GET and PUT requests. The block driver, as we saw, is the component in charge of performing Salus' scalable end-to-end verification (see §4.3): for PUT requests it generates the appropriate metadata, while for GET requests it uses the request's metadata to check whether the data returned to the client is consistent.

To issue a request, the client (or rather, its block driver) contacts the Master, which identifies the RRS responsible for servicing the block that the client wants to access. The client caches this information for future use and forwards the request to that RRS. The first responsibility of the RRS is to ensure that the request commits in the order specified by the client. This is where the pipelined commit protocol becomes important: as we will see in more detail in §4.1, the protocol requires only minimal coordination to enforce dependencies among requests assigned to distinct RRSs. If the request is a PUT, the RRS also needs to ensure that the data associated with the request is made persistent, despite the possibility of individual region servers suffering commission failures. This is the role of active storage (see §4.2): the responsibility of processing PUT requests is no longer assigned to a single region server, but is instead conditioned on the set of region servers in the RRS achieving unanimous consent on the update to be performed. Thanks to Salus' end-to-end verification guarantees, GET requests can instead be safely carried out by a single region server (with obvious performance benefits), without running the risk that the client sees incorrect data.

4.1 Pipelined commit

The goal of the pipelined commit protocol is to allow clients to concurrently issue requests to multiple regions, while preserving the ordering specified by the client (ordered-commit). In the presence of even simple crash failures, however, enforcing the ordered-commit property can be challenging.

Consider, for example, a client that, after mounting a

volume V that spans regions 1 and 2, issues a PUT u_1 for a block mapped to region 1 and then, without waiting for the PUT to complete, issues a barrier PUT u_2 for a block mapped at region 2. Untimely crashes, even transient ones, of the client and of the region server for region 1 may lead to u_1 being lost even as u_2 commits.[3] Volume V now violates both standard disk semantics and the weaker prefix semantics; further, V is left in an invalid state that can potentially cause severe data loss [15, 35].

A simple way to avoid such inconsistencies would be to allow clients to issue one request (or one batch of requests) at a time, but, as we show in §5.2.4, performance would suffer significantly. Instead, we would like to achieve the good performance that comes with issuing multiple oustanding requests, without compromising the ordered-commit property. To achieve this goal, Salus parallelizes the bulk of the processing (such as cryptographic checks and disk-writes) required to handle each request, while ensuring that requests commit in order.

Salus ensures ordered-commit by exploiting the sequence number that clients assign to each request. Region servers use these sequence numbers to guarantee that a request does not commit unless the previous request is also guaranteed to eventually commit. Similarly, during recovery, these sequence numbers are used to ensure that a consistent prefix of issued requests are recovered (§4.4).

Salus' apporach to ensure ordered-commit for GETs is simple. Like other systems before it [9], Salus neither assigns new sequence numbers to GETs, nor logs GETs to stable storage. Instead, to prevent returning stale values, a GET request to a region server simply carries a `prevNum` field indicating the sequence number of the last PUT executed on that region: region servers do not execute a GET until they have committed a PUT with the `prevNum` sequence number. Conversely, to prevent the value of a block from being overwritten by a later PUT, clients block PUT requests to a block that has outstanding GET requests.[4]

Salus' pipelined commit protocol for PUTs is illustrated in Figure 2. The client, as in HBase, issues requests in batches. Unlike HBase, each client is allowed to issue multiple outstanding batches. Each batch is committed using a 2PC-like protocol [21, 24], consisting of the phases described below. Compared to 2PC, pipelined commit reduces the overhead of the failure-free case by eliminating the disk write in the commit phase and by pushing complexity to the recovery protocol, which is usually a good trade-off.

PC1. *Choosing the batch leader and participants.* To process a batch, a client divides its PUTs into various sub-batches, one per region server. Just like a GET request, a PUT request to a region also includes a `prevNum` field to identify the last PUT request issued to that region. The client identifies one region server as *batch leader* for the batch and sends each sub-batch to the appropriate region server along with the batch leader's identity. The client sends the sequence numbers of all requests in the batch to the batch leader, along with the identity of the leader of the previous batch.

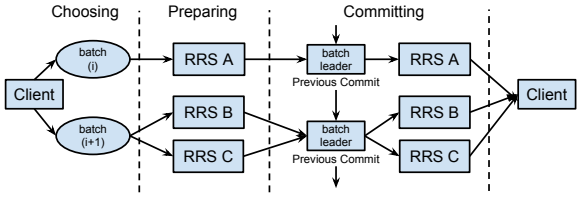


Fig. 2: Pipelined commit (each batch leader is actually replicated to tolerate arbitrary faults.)

PC2. *Preparing.* A region server preprocesses the PUTs in its sub-batch by *validating* each request, i.e. by checking whether the request is signed and by using the `prevNum` field to verify it is the next request that the region server should process. If validation succeeds for all requests in the sub-batch, the region server logs the request (which is now *prepared*) and sends its YES vote to the batch's leader; otherwise, the region server votes NO.

PC3. *Deciding.* The batch leader can decide COMMIT only if it receives a YES vote for all the PUTs in its batch and a COMMIT-CONFIRMATION from the leader of the previous batch; otherwise, it decides ABORT. Either way, the leader notifies the participants of its decision. Upon receiving COMMIT for a request, a region server updates its memory state (memstore), sends a PUT_SUCCESS notification to the client, and asynchronously marks the request as committed on persistent storage. On receiving ABORT, a region server discards the state associated with that PUT and sends the client a PUT_FAILURE message.

Notice that all disk writes—both within a batch and across batches—can proceed in parallel and that the voting and commit phases for a given batch can be similarly parallelized. Different region servers receive and log the PUT and COMMIT asynchronously. The only serialization point is the passing of COMMIT-CONFIRMATION from the leader of a batch to the leader of the next batch.

Despite its parallelism, the protocol ensures that requests commit in the order specified by the client. The presence of COMMIT in any correct region server's log implies that all preceding PUTs in this batch must have prepared. Furthermore, all requests in preceding batches must have also prepared. Our recovery protocol (§4.4) ensures that all these prepared PUTs eventually commit without violating ordered-commit.

The pipelined commit protocol enforces ordered-

[3] For simplicity, in this example and throughout this section we consider a single logical region server to be at work in each region. In practice, in Salus this abstraction is implemented by a RRS.

[4] This requirement has minimal impact on performance, as such PUT requests are rare in practice.

commit assuming the abstraction of (logical) region servers that are correct. It is the *active storage* protocol (§4.2) that, from physical region servers that *can* lose committed data and suffer arbitrary failures, provides this abstraction to the pipelined commit protocol.

4.2 Active storage

Active storage provides the abstraction of a region server that does not experience arbitrary failures or lose data. Salus uses active storage to ensure that the data remains available and durable despite arbitrary failures in the storage system by addressing a key limitation of existing scalable storage systems: they replicate data at the storage layer (e.g. HDFS) but leave the computation layer (e.g. HBase) unreplicated. As a result, the computation layer that processes clients' requests represents a single point of failure in an otherwise robust system. For example, a bug in computing the checksum of data or a corruption of the memory of a region server can lead to data loss and data unavailability in systems like HBase.

The design of Salus embodies a simple principle: all changes to persistent state should happen with the consent of a quorum of nodes. Salus uses these *compute quorums* to protect its data from faults in its region servers.

Salus implements this basic principle using *active storage*. In addition to storing data, storage nodes in Salus also coordinate to attest data and perform checks to ensure that only correct and attested data is being replicated. Perhaps surprisingly, in addition to improving fault-resilience, active storage also enables us to improve performance by trading relatively cheap CPU cycles for expensive network bandwidth.

Using active storage, Salus can provide strong availability and durability guarantees: a data block with a quorum of size n will remain available and durable as long as no more than $n-1$ nodes fail. These guarantees hold irrespective of whether the nodes fail by crashing (omission) or by corrupting their disk, memory, or logical state (commission).

Replication typically incurs network and storage overheads. Salus uses two key ideas—(1) moving computation to data, and (2) using unanimous consent quorums—to ensure that active storage does not incur more network cost or storage cost compared to existing approaches that do not replicate computation.

4.2.1 Moving computation to data to minimize network usage

Salus implements active storage by blurring the boundaries between the storage layer and the compute layer. Existing storage systems [6, 11, 14] require a designated primary datanode to mediate updates. In contrast, Salus modifies the storage system API to permit region servers to directly update any replica of a block. Using this modified interface, Salus can efficiently implement active storage by colocating a compute node (region server) with the storage node (datanode) that it needs to access.

Active storage thus reduces bandwidth utilization in exchange for additional CPU usage (§5.2.2)—an attractive trade-off for bandwidth starved data-centers. In particular, because a region server can now update the colocated datanode without requiring the network, the bandwidth overheads of flushing (§3) and compaction (§3) in HBase are avoided.

We have implemented active storage in HBase by changing the NameNode API for allocating blocks. As in HBase, to create a block a region server sends a request to the NameNode, which responds with the new block's location; but where the HBase NameNode makes its placement decisions in splendid solitude, in Salus the request to the NameNode includes a list of preferred datanodes as a *location-hint*. The hint biases the NameNode toward assigning the new block to datanodes hosted on the same machines that also host the region servers that will access the block. The NameNode follows the hint unless doing so violates its load-balancing policies.

Loosely coupling in this way the region servers and datanodes of a block yields Salus significant network bandwidth savings (§5.2.2): why then not go all the way—eliminate the HDFS layer and have each region server store its state on its local file system? The reason is that maintaining flexibility in block placement is crucial to the robustness of Salus: our design allows the NameNode to continue to load balance and re-replicate blocks as needed, and makes it easy for a recovering region server to read state from any datanode that stores it, not just its own disk.

4.2.2 Using unanimous consent to reduce replication overheads

To control the replication and storage overheads, we use unanimous consent quorums for PUTs. Existing systems replicate data to three nodes to ensure durability despite two permanent omission failures. Salus provides the same durability and availability guarantees despite two failures of either omission *or* commission without increasing the number of replicas. To tolrate f commission faults with just $f+1$ replicas, Salus requires the replicas to reach unanimous consent prior to performing any operation that updates the state and to store a certificate proving the legitimacy of the update.

Of course, the failure of any of the replicated region servers can prevent unanimous consent. To ensure liveness, Salus replaces any RRS that is not making adequate progress with a new set of region servers, which read all state committed by the previous region server quorum from the datanodes and resume processing requests. This fail-over protocol is a slight variation of the one already present in HBase to handle failures of unreplicated region servers. If a client detects a problem with a RRS, it sends a *RRS-replacement request* to the Master, which

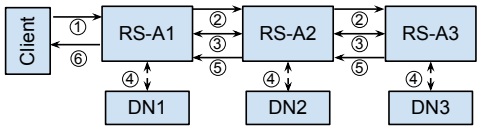


Fig. 3: Steps to process a PUT request in Salus using active storage.

first attempts to get all the nodes of the existing RRS to relinquish their leases; if that fails, the Master coordinates with ZooKeeper to prevent lease renewal. Once the previous RRS is known to be disabled, the Master appoints a new RRS. Then Salus performs the recovery protocol as described in §4.4.

4.2.3 Active storage protocol

To provide to the other components of Salus the abstraction of a correct region server, region servers within a RRS are organized in a chain. In response to a client's PUT request or to attend a periodic task (such as flushing and compaction), the *primary* region server (the first replica in the chain) issues a *proposal*, which is forwarded to all region servers in the chain. After executing the request, the region servers in the RRS coordinate to create a *certificate* attesting that all replicas executed the request in the same order and obtained identical responses. The components of Salus (such as client, NameNode, and Master) that use active storage to make data persistent require all messages from a RRS to carry such a certificate: this guarantees no spurious changes to persistent data as long as at least one region server and its corresponding datanode do not experience a commission failure.

Figure 3 shows how active storage refines the pipelined commit protocol for PUT requests. The PUT issued by a client is received by the primary region server as part of a sub-batch (①). Upon receiving a PUT, each replica validates it and forwards it down the chain of replicas (②). The region servers then agree on the location and order of the PUT in the append-only logs (③) and create a PUT-*log certificate* that attests to that location and order. Each region server sends the PUT and the certificate to its corresponding datanode to guarantee their persistence and waits for the datanode's confirmation (④) before marking the request as prepared. Each region server then independently contacts the leader of the batch to which the PUT belongs and, if it voted YES, waits for the decision. On receiving COMMIT, the region servers mark the request as committed, update their in-memory state and generate a PUT_SUCCESS certificate (⑤); on receiving ABORT the region servers generate instead a PUT_FAILED certificate. In either case, the primary then forwards the certificate to the client (⑥).

Similar changes are also required to leverage active storage in flushing and compaction. Unlike PUTs, these operations are initiated by the primary region server: the other region servers use predefined deterministic criteria, such as the current size of the memstore, to verify whether the proposed operation should be performed.

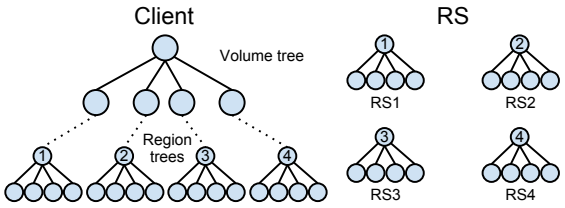


Fig. 4: Merkle tree structure on client and region servers

4.3 End-to-end verification

Local file systems fail in unpredictable ways [35]. Distributed systems like HBase are even more complex and are therefore more prone to failures. To provide strong correctness guarantees, Salus implements end-to-end checks that (a) ensure that clients access correct and current data and (b) do so without affecting performance: GETs can be processed at a single replica and yet retain the ability to identify whether the returned data is correct and current.

Like many existing systems [19, 26, 31, 41], Salus' mechanism for end-to-end checks leverages Merkle trees to efficiently verify the integrity of the state whose hash is at the tree's root. Specifically, a client accessing a volume maintains a Merkle tree on the volume's blocks, called *volume tree*, that is updated on every PUT and verified on every GET.

For robustness, Salus keeps a copy of the volume tree stored distributedly across the region servers that host the volume so that, after a crash, a client can rebuild its volume tree by contacting the region servers responsible for the regions in that volume. Replicating the volume tree at the region servers also allows a client, if it so chooses, to only store a subset of its volume tree during normal operation, fetching on demand what it needs from the region servers serving its volume.

Since a volume can span multiple region servers, for scalability and load-balancing each region server only stores and validates a *region tree* for the regions that it hosts. The region tree is a sub-tree of the volume tree corresponding to the blocks in a given region. In addition, to enable the client to recover the volume tree, each region server also stores the latest known hash for the root of the full volume tree, together with the sequence number of the PUT request that produced it.

Figure 4 shows a volume tree and its region trees. The client stores the top levels of the volume tree that are not included in any region tree so that it can easily fetch the desired region tree on demand. A client can also cache recently used region trees for faster access.

To process a GET request for a block, the client sends the request to any of the region servers hosting that block. On receiving a response, the client verifies it using the locally stored volume tree. If the check fails (because of a commission failure) or if the client times out (because of

an omission failure), the client retries the GET using another region server. If the GET fails at all region servers, the client contacts the Master triggering the recovery protocol(§4.4). To process a PUT, the client updates its volume tree and sends the weakly-signed root hash of its updated volume tree along with the PUT request to the RRS. Attaching the root hash of the volume tree to each PUT request enables clients to ensure that, despite commission failures, they will be able to mount and access a consistent volume.

A client's protocol to mount a volume after losing the volume tree is simple. The client begins by fetching the region trees, the root hashes, and the corresponding sequence numbers from the various RRSs. Before responding to a client's fetch request, a RRS commits any prepared PUTs pending to be committed using the *commit-recovery* phase of the recovery protocol (§4.4). Using the sequence numbers received from all the RRSs, the client identifies the most recent root hash and compares it with the root hash of the volume tree constructed by combining the various region trees. If the two hashes match, then the client considers the mount to be complete; otherwise it reports an error indicating that a RRS is returning a potentially stale tree. In such cases, the client reports an error to the Master to trigger the replacement of the servers in the corresponding RRS, as described in §4.4.

4.4 Recovery

The recovery protocol ensures that, despite comission or permanent omission failures in up to f pairs of corresponding region servers and datanodes, Salus continues to provide the abstraction of a virtual disk with *standard disk semantics*, except in the extreme failure scenario in which the client crashes *and* one or more of the region server/datanode pairs of a region experience a commission failure *and* all other region server/datanode pairs of that region are unavailable: in this case, Salus' recovery protocol guarantees the weaker *prefix semantics*.

To achieve this goal, Salus' recovery protocol collects the longest available prefix P_C of prepared PUT requests that satisfy the ordered-commit property. Recall from §4.1 that every PUT for which the client received a PUT_SUCCESS must appear in the log of at least one correct replica in the region that processed that PUT. Hence, if a correct replica is available for each of the volume's regions, P_C will contain all PUT requests for which the client received a PUT_SUCCESS, thus guaranteeing standard disk semantics. If however, because of a confluence of commission and transient omission failures, the only available replicas in a region are those who have suffered commission failures, then the P_C that the recovery protocol collects may include only a prefix (albeit ordered-commit-compliant) of those PUT requests, resulting in the weaker prefix semantics.

Specifically, recovery must address two key issues.

Resolving log discrepancies Because of omission or commission failures, different datanodes within the same RRS may store different logs. A prepared PUT, for example, may have been made persistent at one datanode, but not at another.

```
1  do
2   foreach failed-region i
3    remapRegion(i)
4   end
5   foreach failed-region i
6     region_logs[i] ← recoverRegionLog(region i)
7   end
8   LCP ← identifyLCP(region_logs)
9  while rebuildVolume(LCP) fails
```

Fig. 5: Pseudocode for the recovery protocol.

Identifying committable requests Because COMMIT decisions are logged asynchronously, some PUTs for which a client received PUT_SUCCESS may not be marked as committed in the logs. It is possible, for example, that a later PUT be logged as committed when an earlier one is not; or that a suffix of PUTs for which the client has received a PUT_SUCCESS be not logged as committed. Worse, because of transient omission failures, some region may temporarily have no operational correct replica when the recovery protocol attempts to collect logged PUTs.

One major challenge in addressing these issues is that, while P_C is defined on a global *volume log*, Salus does not actually store any such log: instead, for efficiency, each region keeps its own separate *region log*. Hence, after retrieving its region log, a recovering region server needs to cooperate with other region servers to determine whether the recovered region log is correct and whether the PUTs it stores can be committed.

Figure 5 describes the protocol that Salus uses to recover faulty datanodes and region servers. The first two phases describe the recovery of individual region logs, while the last two phases describe how the RRSs coordinate to identify commitable requests.

1. Remap (`remapRegion`). As in HBase, when a RRS crashes or is reported by the client as non-responsive, the Master swaps out the servers in that RRS and assigns its regions to one or more replacement RRSs.

2. Recover region log (`recoverRegionLog`). To recover all prepared PUTs of a failed region, the new region servers choose, among the instances (one for each operational datanode) of that region's old region logs, the longest available log that is *valid*. A log is *valid* if it is a prefix of PUT requests issued to that region. [5] We use the PUT-*log certificate* attached to each PUT record to separate *valid* logs from *invalid* ones: each region server independently replays the log and checks if each PUT record's location and order matches the location and order included in that PUT record's PUT-*log certificate*. Having found a valid log, the servers in the RRS agree on the longest prefix and advance to the next stage.

[5] Salus' approach for truncating logs is similar to how HBase manages checkpoints and is discussed in an extended TR [44].

3. Identify the longest commitable prefix (LCP) of the volume log (`identifyLCP`). If the client is available, Salus can determine the LCP and the root of the corresponding volume tree simply by asking the client. Otherwise, all RRSs must coordinate to identify the longest prefix of the volume log that contains either committed or prepared PUTs (i.e. PUTs whose data has been made persistent in at least one correct datanode). Since Salus keeps no physical volume log, the RRSs use ZooKeeper as a means of coordination, as follows. The Master asks each RRS to report its maximum committed sequence number as well as its list of prepared sequence numbers by writing the requested information to a known file in ZooKeeper. Upon learning from Zookeeper that the file is complete (i.e. all RRSs have responded), [6] each RRS uses the file to identify the longest prefix of committed and prepared PUTs in the volume log. Finally, the sequence number of the last PUT in the LCP and the attached Merkle tree root are written to ZooKeeper.

4. Rebuild volume state (`rebuildVolume`). The goal of this phase is to ensures that all PUTs in the LCP are committed and available. The first half is simple: if a PUT in the LCP is prepared, then the corresponding region server marks it as committed. With respect to availability, Salus makes sure that all PUTs in the LCP are available, in order to reconstruct the volume consistently. To that end, the Master asks the RRSs to replay their log and rebuild their region trees; it then uses the same checks used by the client in the mount protocol (§4.3) to determine whether the current root of the volume tree matches the one stored in ZooKeeper during Phase 3.

As mentioned above, a confluence of commission and transient omission failures could cause a RRS to recover only a prefix of its region log. In that case, the above checks could fail, since some PUTs within the LCP could be ignored. If the checks fail, recovery starts again from Phase 1.[7] Note, however, that all the ignored PUTs must have been prepared and so, as long as the number of permanent omission or commission failures does not exceed f, a correct datanode will eventually become available and a consistent volume will be recovered.

5 Evaluation

We have implemented Salus by modifying HBase [6] and HDFS [39] to add pipelined commit, active storage, and end-to-end checks. Our current implementation lags behind our design in two ways. First, our prototype supports unanimous consent between HBase and HDFS but not between HBase and ZooKeeper. Second, while our design calls for a BFT-replicated Master, NameNode, and ZooKeeper, our prototype does not yet incorporate these features. We intend to use UpRight [16] to replicate NameNode, ZooKeeper, and Master.

[6] If some RRS are unavailable during this phase, recovery starts again from Phase 1, replacing the unavailable servers.

[7] For a more efficient implementation that leverages version vectors, see [44].

Salus ensures *freshness, ordered-commit, and liveness* when there are no more than 2 failures within any RRS and the corresponding datanodes.	§5.1
Salus achieves comparable or better single-client throughput compared to HBase with slightly increased latency.	§5.2.1
Salus' active replication can reduce network usage by 55% and increase aggregate throughput by 74% for sequential write workload compared to HBase. Salus can achieve similar aggregate read throughput compared to HBase.	§5.2.2
Salus' overhead over HBase does not grow with the scale of the system.	§5.2.3

Fig. 6: Summary of main results.

Our evaluation tries to answer two basic questions. First, does Salus provide the expected guarantees despite a wide range of failures? Second, given its stronger guarantees, is Salus' performance competitive with HBase? Figure 6 summarizes the main results.

5.1 Robustness

In this section, we evaluate Salus' robustness, which includes guaranteeing freshness for read operations and liveness and ordered-commit for all operations.

Salus is designed to ensure these properties as long as there are no more than two failures in the region servers within an RRS and their corresponding datanodes, and fewer than a third of the nodes in the implementation of each of UpRight NameNode, UpRight ZooKeeper, and UpRight Master nodes are incorrect; however, since we have not yet integrated in Salus UpRight versions of NameNode, ZooKeeper, and Master, we only evaluate Salus' robustness when datanode or region server fails.

We test our implementation via fault injection. We introduce failures and then determine what happens when we attempt to access the storage. For reference, we compare Salus with HBase (which replicates stored data across datanodes but does not support pipelined commit, active storage, or end-to-end checks).

In particular, we inject faults into clients to force them to crash and restart. We inject faults into datanodes to force them either to crash, temporarily or permanently, or to corrupt block data. We cause data corruption in both log files and checkpoint files. We inject faults into region servers to force them to either 1) crash; 2) corrupt data in memory; 3) write corrupted data to HDFS; 4) refuse to process requests or forward requests out of order; or 5) ask the NameNode to delete files. Once again, we cause corruption in both log files and checkpoint files. Note that data on region servers is not protected by checksums. Figure 7 summarizes our results.

First, as expected, when a client crashes and restarts in HBase, a volume's on-disk state can be left in an inconsistent state, because HBase does not guarantee ordered commit. HBase can avoid these inconsistencies by blocking all requests that follow a barrier request until the barrier completes, but this can hurt performance

Affected nodes	Faults	HBase		Salus	
		GET	PUT	GET	PUT
Client	Crash and restart	Fresh	Not ordered	Fresh	Ordered
DataNode	1 or 2 permanent crashes	Fresh	Ordered	Fresh	Ordered
	Corruption of 1 or 2 replicas of log or checkpoint	Fresh	Ordered	Fresh	Ordered
	3 arbitrary failures	Fresh*	Lost	Fresh*	Lost
Region server+DataNode	1 (for HBase) or 3 (for Salus) region server permanent crashes	Fresh	Ordered	Fresh	Ordered
	1 (for HBase) or 2 (for Salus) region server arbitrary failures that potentially affect datanodes	Corrupted	Lost	Fresh	Ordered
	3 (for Salus) region server arbitrary failures that potentially affect datanodes	-	-	Fresh*	Lost
Client+Region server+DataNode	Client crashes and restarts, 1 (for HBase) or 2 (for Salus) region server arbitrary failures causing the corresponding datanodes to not receive a suffix of data	Corrupted	Lost	Fresh	Ordered

Fig. 7: Robustness towards failures affecting the region servers within an RRS, and their corresponding datanodes. (- = not applicable, * = corresponding operations may not be live). Note that a region server failure has the potential to cause the failure of the corresponding datanode.

when barriers are frequent (see §5.2.4). Second, HBase's replicated datanodes tolerate crash and *benign* file corruptions that alter the data but don't affect the checksum, which is stored separately. Thus, when considering only datanode failures, HBase provides the same guarantees as Salus. Third, HBase's unreplicated region server is a single point of failure, vulnerable to commission failures that can violate freshness as well as ordered-commit.

In Salus, end-to-end checks ensure freshness for GET operations in all the scenarios covered in Figure 7: a correct client does not accept GET reply unless it can pass the Merkle tree check. Second, pipelined commit ensures the ordered-commit property in all scenarios involving one or two failures, whether of omission or of commission: if a client fails or region servers reorder requests, the out-of-order requests will not be accepted and eventually recovery will be triggered, causing these requests to be discarded. Third, active storage protects liveness failure scenarios involving one or two region server/datanode pairs: if a client receives an unexpected GET reply, it retries until it obtains the correct data. Furthermore, during recovery, the recovering region servers find the correct log by using the certificates generated by active storage protocol. As expected, ordered-commit and liveness cannot be guaranteed if *all* replicas either permanently fail or experience commission failures.

5.2 Performance

Salus' architecture can in principle result in both benefits and overhead when it comes to throughput and latency: on the one hand, pipelined commit allows multiple batches to be processed in parallel and active storage reduces network bandwidth consumption. On the other hand, end-to-end checks introduce checksum computations on both clients and servers; pipelined commit requires additional network messages for preparing and committing; and active storage requires additional computation and messages for certificate generation and validation. Compared to the cost of disk-accesses for data, however, we expect these ovrheads to be modest.

This section quantifies these tradeoffs using sequential- and random-, read and write microbenchmarks. We compare Salus' single-client throughput and latency, aggregate throughput, and network usage to those of HBase. We also include measured numbers from Amazon EBS to put Salus' performance in perspective.

Salus targets clusters of storage nodes with 10 or more disks each. In such an environment, we expect a node's aggregate disk bandwidth to be much larger than its network bandwidth. Unfortunately, we have only three *storage nodes* matching this description, the rest of our *small nodes* have a single disk and a single active 1Gbit/s network connection.

Most of our experiments run on a 15-node cluster of *small nodes* equipped with a 4-core Intel Xeon X3220 2.40GHz CPU, 3GB of memory, and one WD2502ABYS 250GB hard drive. In these experiments, we use nine small nodes as region servers and datanodes, another small node as the Master, ZooKeeper, and NameNode, and up to four small nodes acting as clients. In these experiments, we set the Java heap size to 2GB for the region server and 1GB for the datanode.

To understand system behavior when disk bandwidth is more plentiful than network bandwidth, we run several experiments using the three storage nodes, each equipped with an 16-core AMD Opteron 4282 @3.0GHz, 64GB of memory, and 10 WDC WD1003FBYX 1TB hard drives. These storage nodes have 1Gbit/s networks, but the network topology constrains them to share an aggregate bandwidth of about 1.2Gbit/s.

To measure the scalability of Salus with a large number of machines, we run several experiments on Amazon EC2 [2]. The detailed configuration is shown in §5.2.3.

For all experiments, we use a small 4KB block size, which we expect to magnify Salus' overheads compared to larger block sizes. For read workloads, each client formats the volume by writing all blocks and then forcing a flush and compaction before the start of the experiments. For write workloads, since compaction introduces significant overhead in both HBase and Salus and the compaction interval is tunable, we first run these

experiments with compaction disabled to measure the maximum throughput; then we run HBase with its default compaction strategy and measure how many bytes it reads for each compaction; finally, we tune Salus' compaction interval so that Salus performs compaction on the same amount of data as HBase.

5.2.1 Single client throughput and latency

We first evaluate the single-client throughput and latency of Salus. Since a single client usually cannot saturate the system, we find that executing requests in a pipeline is beneficial to Salus' throughput. However, the additional overhead of checksum computation and message transfer of Salus increases its latency.

We use the nine small nodes as servers and start a single client to issue sequential and random reads and writes to the system. For the throughput experiment, the client issues requests as fast as it can and performs batching to maximize throughput. In all experiments, we use a batch size of 250 requests, so each batch accesses about 1MB. For the latency experiment, the client issues a single request, waits for it to return, and then waits for 10ms before issuing the next request.

Figure 8 shows the single client throughput. For sequential read, Salus outperforms the HBase system with a speedup of 2.5. The reasons are that Salus' active replication's three region servers increase parallelism for reads and reads are pipelined to have multiple batches outstanding; the HBase client instead issues only one batch of requests at a time. For random reads, disk seeks are the bottleneck and HBase and Salus have comparable performance.

For sequential write and random write, Salus is slower than HBase by 3.5% to 22.8% for its stronger guarantees. For Salus, pipelined execution does not help write throughput as much as it helps sequential reads, since write operations need to be forwarded to all three nodes and unlike reads cannot be executed in parallel.

As a sanity check, Figure 8 also shows the performance we measured from a small compute instance accessing Amazon's EBS. Because the EBS hardware differs from our testbed hardware, we can only draw limited conclusions, but we note that the Salus prototype achieves a respectable fraction of EBS's sequential read and write bandwidth, and that it modestly outperforms EBS's random read throughput (likely because it is utilizing more disk arms), and that it substantially outperforms EBS's random write throughput (likely because it transforms random writes into sequential ones.)

Figure 9 shows the 90th-percentile latency for random reads and writes. In both cases, Salus' latency is within two or three milliseconds or of HBase's. This is reasonable considering Salus' additional work to perform Merkle tree calculation, certificate generation and validation, and network transfer. One thing should be noted

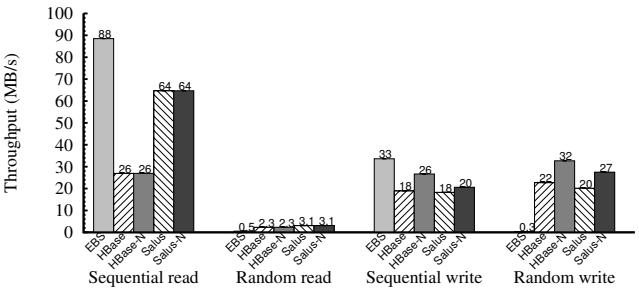


Fig. 8: Single client throughput on small nodes. HBase-N and Salus-N disable compactions. EBS's numbers are measured on different hardwares and are included for reference.

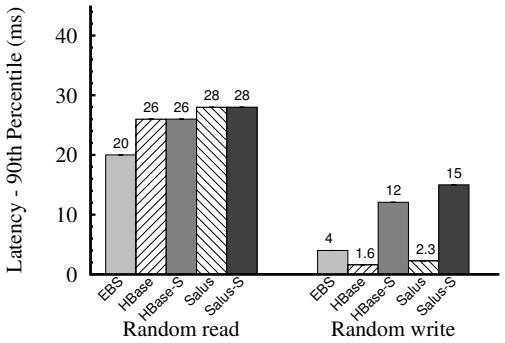


Fig. 9: Single client latency on small nodes. HBase-S and Salus-S enable sync. EBS's numbers are measured on different hardwares and are included for reference.

about the random write latency experiment: the HBase datanode does not call *sync* when performing disk write and that's why its write latency is small. This may be a reasonable design decision when the probability of three simultaneous crashes is small [28]. In this experiment, we also show what happens when adding this call to both HBase and Salus: calling sync adds more than 10ms of latency to both. To be consistent, we do not call *sync* in other throughput experiments.

Again, as a sanity check we note that Salus (and HBase) are reasonably competitive with EBS (though we emphasize again that EBS's underlying hardware is not known to us, so not too much should be read into this experiment.)

Overall, these results show that despite all the extra computation and message transfers to achieve stronger guarantees, Salus' single-client throughput and latency are comparable to those of HBase. This is because the additional processing Salus requires adds relatively little to the time required to complete disk operations. In an environment in which computational cycles are plentiful, trading off as Salus does processing for improved reliability appears to be a reasonable trade-off.

5.2.2 Aggregate throughput/network bandwidth

We then evaluate the aggregate throughput and network usage of Salus. The servers are saturated in these experiments, so pipelined execution does not improve Salus' throughput at all. On the other hand, we find that active

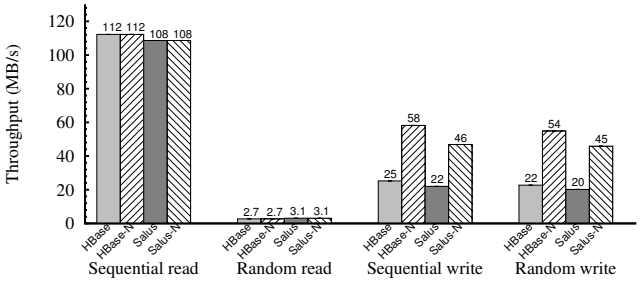


Fig. 10: Aggregate throughput on small nodes. HBase-N and Salus-N disable compactions.

	HBase	Salus
Throughput (MB/s)	27	47
Network consumption (network bytes per byte written by the client)	5.3	2.4

Fig. 11: Aggregate sequential write throughput and network bandwidth usage with fewer server machines but more disks per machine.

replication of region servers, introduced to improve robustness, can reduce network bandwidth and significantly improve performance when the total disk bandwidth exceeds the aggregate network bandwidth.

Figure 10 reports experiments on our small-server testbed with nine nodes acting as combined region server and datanode servers and we increase the number of clients until the throughput does not increase.

For sequential read, both systems can achieve about 110MB/s. Pipelining reads in Salus does not improve aggregate throughput since also HBase has multiple clients to parallelize network and disk operations. For random reads, disk seek and rotation are the bottleneck, and both systems achieve only about 3MB/s.

The relative slowdown of Salus versus HBase for sequential and random writes is respectively of 11.1% to 19.4% and significantly lower when compaction is enabled since compaction adds more disk operations to both HBase and Salus. Salus reduces network bandwidth at the expense of higher disk and CPU usage, but this trade-off does not help in our system because disk and network bandwidth are comparable. Even so, we find this to be an acceptable price for the stronger guarantees provided by Salus.

Figure 11 shows what happens when we run the sequential write experiment using the three 10-disk storage nodes as servers. Here, the tables are turned and Salus outperforms HBase (47MB/s versus 27MB/s). Our profiling shows that in both experiments, the bottleneck is the network topology that constrains the aggregate bandwidth to 1.2Gbit/s.

Figure 11 also compares the network bandwidth usage of HBase and Salus under the sequential write workload. HBase sends more than five bytes for each byte written by the client (two network transfers each for logging and flushing, but fewer than two for compaction, since some blocks are overwritten.) Salus only uses two bytes per-byte-written to forward the request to replicas;

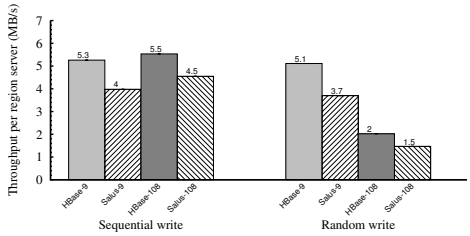


Fig. 12: Write throughput per server with 9 servers and 108 servers (compaction disabled).

logging, flushing, and compaction are performed locally. The actual number is slightly higher than 2, because of Salus's additional metadata. Salus halves network bandwidth usage compared to HBase, which explains why its throughput is 74% higher than that HBase when network bandwidth is limited.

Note that we do not measure the aggregate throughput of EBS because we do not know its internal architecture and thus we do not know how to saturate it.

5.2.3 Scalability

In this section we evaluate the degree to which the mechanisms that Salus uses to achieve its stronger robustness guarantees impact its scalability. Growing by an order of magnitude the size of the testbed used in our previous experiments, we run Salus and HBase on Amazon EC2 [2] with up to 108 servers. While short of our goal of showing conclusively that Salus can scale to thousands of servers, we believe these experiments can offer valuable insights on the relevant trends.

For our testbed we use EC2's extra large instances, with datanodes and region servers configured to use 3GB of memory each. Some preliminary tests run to measure the characteristics of our testbed show that each EC2 instance can reach a maximum network and disk bandwidth of about 100MB/s, meaning that network bandwidth is not a bottleneck; thus, we do not expect Salus to outperform HBase in this setting.

Given our limited resources, we focus our attention on measuring the throughput of sequential and random writes: we believe this is reasonable since the only additional overhead for reads are the end-to-end checks performed by the clients, which are easy to make scalable. We run each experiment with an equal number of clients and servers and for each 11-minute-long experiment we report the throughput of the last 10 minutes.

Because we do not have full control over EC2's internal architecture, and because one user's virtual machines in EC2 may share resources such as disks and networks with other users, these experiments have limitations: the performance of EC2's instances fluctuates noticeably and it becomes hard to even determine what the stable throughput for a given experimental configuration is. Further, while in most cases, as expected, we find that HBase performs better than Salus, some experi-

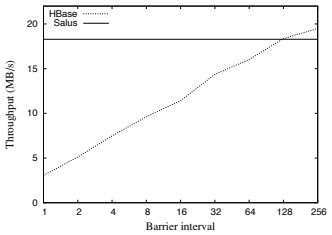


Fig. 13: Single client sequential write throughput as the frequency of barriers varies.

ments show Salus with a higher throughput than HBase, possibly because the network is being heavily used and pipelined commit helps Salus handle high network latencies more efficiently: to be conservative, we report only results for which HBase performs better than Salus.

Figure 12 shows the per-server throughput of the sequential and random write workloads in configuration with 9 and 108 servers. For the sequential write workload, the throughput per server remains almost unchanged in both HBase and Salus as we move from 9 to 108 servers, meaning that for this workload both systems are perfectly scalable up to 108 servers. For the random write workload, however, both HBase and Salus experience a significant drop in throughput-per-server when the number of servers grows. The culprit is the high number of small I/O operations that this workload requires. As the number of server increases, the number of requests randomly assigned to each server in a sub-batch decreases, even as increasing the number of clients causes each server to process more sub-batches. The net result is that as the number of server increases, each server performs an ever larger number of ever smaller-sized I/O operations—which of course hurts performance. Note however that the extent of Salus' slowdown with respect to HBase is virtually the same (28%) in both the 9-server and the 108-server experiments, the letter that Salus' overhead does not grow with the scale of the system.

5.2.4 Pipeline commit

Salus achieves increased parallelism by pipelining PUTs across barrier operations—Salus' PUTs always commit in the order they are issued, so the barriers' constraints are satisfied without stalling the pipeline. Figure 13 compares HBase and Salus by varying the number of operations between barriers. Salus' throughput remains constant at 18 MB/s as it is not affected by barriers, whereas HBase's throughput suffers with increasing barrier frequency: HBase achieves 3MB/s with a batch size of 1 and 14 MB/s with a batch size of 32.

6 Related work

Scalable and consistent storage. Many existing systems provide the abstraction of scalable distributed storage [6, 11, 14, 27] with strong consistency. Unfortunately, these systems do not tolerate arbitrary node failures. While these systems use checksums to safeguard data written on disk, a memory corruption or a software glitch can lead to the loss of data in these systems (§ 5.1). In contrast, Salus is designed to be robust (safe and live) even if nodes fail arbitrarily.

Protections in local storage systems Disks and storage sub systems can fail in various ways [7, 8, 18, 23, 34, 37], are so can memories and CPUs [33, 38] with disastrous consequences [35]. Unfortuantely, end-to-end protection mechanisms developed for local storage systems [35, 41] are inadequate for protecting the full path from a PUT to a GET in complex systems like HBase.

End-to-end checks. ZFS [41] incorporates an on-disk Merkle tree to protect the file system from disk corruptions. SFSRO [19], SUNDR [26], Depot [31], and Iris [40] also use end-to-end checks to guard against faulty servers. However, none of these systems is designed to scale to thousands of machines, because, to support multiple clients sharing a volume, they depend on a single server to update the Merkle tree. Instead, Salus is designed for a single client per volume, so it can rely on the client to update the Merkle tree and make the server side scalable. We do not claim this to be a major novelty of Salus; we see this as an example of how different goals lead to different designs.

BFT systems. While some distributed systems tolerate arbitrary faults (Depot [31], SPORC [17], SUNDR [26], BFT RSM [13, 16]), they require a correct node to observe all writes to a given volume, preventing a volume from scaling with the number of nodes.

Supporting multiple writers. We are not aware of any system that can support multiple writers while achieving ordered-commit, scalability, and end-to-end verification for read requests. One can tune Salus to support multiple writers by either using a single server to serialize requests to a volume as shown in SUNDR [26], which of course hurts scalability, or by using weaker consistency models like Fork-Join-Casual [31] or fork* [17].

7 Conclusions

Salus is a distributed block store that offers an unprecedented combination of scalability and robustness. Surprisingly, Salus' robustness does not come at the cost of performance: pipelined commit allows updates to proceed at high speed while ensuring that the system's committed state is consistent; end-to-end checks allow reading from one replica safely; and active replication not only eliminates reliability bottlenecks but also eases performance bottlenecks.

Acknowledgements

We thank our shepherd Arvind Krishnamurthy and the anonymous reviewers for their insightful comments. This work was supported in part by NSF grants CiC-FRCC-1048269 and CSR-0905625.

References

[1] Amazon Elastic Block Store (EBS). http://aws.amazon.com/ebs/.

[2] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/.

[3] D. Anderson, J. Chase, and A. Vahdat. Interposed Request Routing for Scalable Network Storage. In *OSDI*, 2000.

[4] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. *ACM Transactions on Computer Systems*, 14(1):41–79, Feb 1996.

[5] Apache Hadoop FileSystem and its Usage in Facebook. http://cloud.berkeley.edu/data/hdfs.pdf.

[6] Apache HBASE. http://hbase.apache.org/.

[7] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder. An analysis of data corruption in the storage stack. *TOS*, 2008.

[8] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. In *SIGMETRICS*, 2007.

[9] W. Bolosky, D. Bradshaw, R. Haagens, N. Kusters, and P. Li. Paxos Replicated State Machines as the Basis of a High-Performance Data Store. In *NSDI*, 2011.

[10] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[11] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[12] M. Castro and B. Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *OSDI*, 2000.

[13] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *TOCS*, 2002.

[14] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[15] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency Without Ordering. In *FAST*, 2012.

[16] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riché. UpRight Cluster Services. In *SOSP*, 2009.

[17] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group Collaboration Using Untrusted Cloud Resources. In *OSDI*, 2010.

[18] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *OSDI*, 2010.

[19] K. Fu, M. F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. *TOCS*, 2002.

[20] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.

[21] J. Gray. Notes on Database Systems. IBM Research Report RJ2188, Feb. 1978.

[22] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

[23] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky. Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics. *TOS*, 2008.

[24] B. Lampson and H. Sturgis. Crash Recovery in a Distributed System. Xerox PARC Research Report, 1976.

[25] E. Lee and R. A. Thekkath. Petal: Distributed virtual disks. In *ASPLOS*, 1996.

[26] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *OSDI*, 2004.

[27] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *SOSP*, 2011.

[28] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, and L. Shrira. Replication in the Harp File System. In *SOSP*, 1991.

[29] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[30] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, 2004.

[31] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: cloud storage with minimal trust. In *OSDI*, 2010.

[32] R. Merkle. Protocols for public key cryptosystems. In *Symposium on Security and Privacy*, 1980.

[33] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer PCs. In *Eurosys*, 2011.

[34] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *FAST*, 2007.

[35] V. Prabhakaran, L. Bairavasundaram, N. Agrawal, H. Gunawi, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. IRON file systems. In *SOSP*, 2005.

[36] R. L. Rivest, A. Shamir, and L. M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems (Reprint). *CACM*, 1983.

[37] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *FAST*, 2007.

[38] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. In *SIGMETRICS*, 2009.

[39] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *MSST*, 2010.

[40] E. Stefanov, M. van Dijk, A. Oprea, and A. Juels. Iris: A Scalable Cloud File System with Efficient Integrity Checks. In *ACSAC*, 2012.

[41] I. Sun Microsystems. ZFS on-disk specification. Technical report, Sun Microsystems, 2006.

[42] HDFS Usage in Yahoo! http://www.aosabook.org/en/hdfs.html.

[43] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *SOSP*, 1997.

[44] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin. Robustness in the Salus scalable block store. Technical Report TR-12-24, The University of Texas at Austin, Department of Computer Science, 2012.

MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing

Bin Fan, David G. Andersen, Michael Kaminsky*
Carnegie Mellon University, *Intel Labs

Abstract

This paper presents a set of architecturally and workload-inspired algorithmic and engineering improvements to the popular Memcached system that substantially improve both its memory efficiency and throughput. These techniques—optimistic cuckoo hashing, a compact LRU-approximating eviction algorithm based upon CLOCK, and comprehensive implementation of optimistic locking—enable the resulting system to use 30% less memory for small key-value pairs, and serve up to 3x as many queries per second over the network. We have implemented these modifications in a system we call MemC3—Memcached with CLOCK and Concurrent Cuckoo hashing—but believe that they also apply more generally to many of today's read-intensive, highly concurrent networked storage and caching systems.

1 Introduction

Low-latency access to data has become critical for many Internet services in recent years. This requirement has led many system designers to serve all or most of certain data sets from main memory—using the memory either as their primary store [19, 26, 21, 25] or as a cache to deflect hot or particularly latency-sensitive items [10].

Two important metrics in evaluating these systems are performance (throughput, measured in queries served per second) and memory efficiency (measured by the overhead required to store an item). Memory consumption is important because it directly affects the number of items that system can store, and the hardware cost to do so.

This paper demonstrates that careful attention to algorithm and data structure design can significantly improve throughput and memory efficiency for in-memory data stores. We show that traditional approaches often fail to leverage the target system's architecture and expected workload. As a case study, we focus on Memcached [19], a popular in-memory caching layer, and show how our toolbox of techniques can improve Memcached's performance by 3× and reduce its memory use by 30%.

Standard Memcached, at its core, uses a typical hash table design, with linked-list-based chaining to handle collisions. Its cache replacement algorithm is strict LRU, also based on linked lists. This design relies on locking to ensure consistency among multiple threads, and leads to poor scalability on multi-core CPUs [11].

This paper presents MemC3 (**Mem**cached with **C**LOCK and **C**oncurrent **C**uckoo Hashing), a complete redesign of the Memcached internals. This re-design is informed by and takes advantage of several observations. First, architectural features can hide memory access latencies and provide performance improvements. In particular, our new hash table design exploits CPU cache locality to minimize the number of memory fetches required to complete any given operation; and it exploits instruction-level and memory-level parallelism to overlap those fetches when they cannot be avoided.

Second, MemC3's design also leverages workload characteristics. Many Memcached workloads are predominately reads, with few writes. This observation means that we can replace Memcached's exclusive, global locking with an optimistic locking scheme targeted at the common case. Furthermore, many important Memcached workloads target very small objects, so per-object overheads have a significant impact on memory efficiency. For example, Memcached's strict LRU cache replacement requires significant metadata—often more space than the object itself occupies; in MemC3, we instead use a compact CLOCK-based approximation.

The specific contributions of this paper include:

- A novel hashing scheme called *optimistic cuckoo hashing*. Conventional cuckoo hashing [23] achieves space efficiency, but is unfriendly for concurrent operations. Optimistic cuckoo hashing (1) achieves high memory efficiency (e.g., 95% table occupancy); (2) allows multiple readers and a single writer to concurrently access the hash table; and (3) keeps hash table operations cache-friendly (Section 3).
- A compact CLOCK-based eviction algorithm that requires only 1 bit of extra space per cache entry and supports concurrent cache operations (Section 4).
- Optimistic locking that eliminates inter-thread syn-

function	stock Memcached	MemC3
Hash Table		
concurrency	serialized	concurrent lookup, serialized insert
lookup performance	slower	faster
insert performance	faster	slower
space	13.3n Bytes	~ 9.7n Bytes
Cache Mgmt		
concurrency	serialized	concurrent update, serialized eviction
space	18n Bytes	n bits

Table 1: Comparison of operations. *n* is the number of existing key-value items.

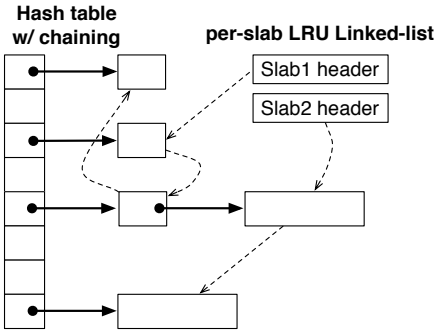


Figure 1: Memcached data structures.

chronization while ensuring consistency. The optimistic cuckoo hash table operations (lookup/insert) and the LRU cache eviction operations both use this locking scheme for high-performance access to shared data structures (Section 4).

Finally, we implement and evaluate MemC3, a networked, in-memory key-value cache, based on Memcached-1.4.13.[1] Table 1 compares MemC3 and stock Memcached. MemC3 provides higher throughput using significantly less memory and computation as we will demonstrate in the remainder of this paper.

2 Background

2.1 Memcached Overview

Interface Memcached implements a simple and lightweight key-value interface where all key-value tuples are stored in and served from DRAM. Clients communicate with the Memcached servers over the network using the following commands:

- `SET/ADD/REPLACE(key, value)`: add a (key, value) object to the cache;
- `GET(key)`: retrieve the value associated with a key;
- `DELETE(key)`: delete a key.

Internally, Memcached uses a hash table to index the key-value entries. These entries are also in a linked list sorted by their most recent access time. The least recently used (LRU) entry is evicted and replaced by a newly inserted entry when the cache is full.

Hash Table To lookup keys quickly, the location of each key-value entry is stored in a hash table. Hash collisions are resolved by chaining: if more than one key maps into the same hash table bucket, they form a linked list. Chaining is efficient for inserting or deleting single keys. However, lookup may require scanning the entire chain.

Memory Allocation Naive memory allocation (e.g., malloc/free) could result in significant memory fragmentation. To address this problem, Memcached uses *slab-based memory allocation*. Memory is divided into 1 MB pages, and each page is further sub-divided into fixed-length *chunks*. Key-value objects are stored in an appropriately-size chunk. The size of a chunk, and thus the number of chunks per page, depends on the particular slab class. For example, by default the chunk size of slab class 1 is 72 bytes and each page of this class has 14563 chunks; while the chunk size of slab class 43 is 1 MB and thus there is only 1 chunk spanning the whole page.

To insert a new key, Memcached looks up the slab class whose chunk size best fits this key-value object. If a vacant chunk is available, it is assigned to this item; if the search fails, Memcached will execute cache eviction.

Cache policy In Memcached, each slab class maintains its own objects in an LRU queue (see Figure 1). Each access to an object causes that object to move to the head of the queue. Thus, when Memcached needs to evict an object from the cache, it can find the least recently used object at the tail. The queue is implemented as a doubly-linked list, so each object has two pointers.

Threading Memcached was originally single-threaded. It uses libevent for asynchronous network I/O callbacks [24]. Later versions support multi-threading but use global locks to protect the core data structures. As a result, operations such as index lookup/update and cache eviction/update are all serialized. Previous work has shown that this locking prevents current Memcached from scaling up on multi-core CPUs [11].

Performance Enhancement Previous solutions [4, 20, 13] shard the in-memory data to different cores. Sharding eliminates the inter-thread synchronization to permit higher concurrency, but under skewed workloads it may also exhibit imbalanced load across different cores or waste the (expensive) memory capacity. Instead of simply

[1] Our prototype does not yet provide the full memcached api.

sharding, we explore how to scale performance to many threads that share and access the same memory space; one could then apply sharding to further scale the system.

2.2 Real-world Workloads: Small and Read-only Requests Dominate

Our work is informed by several key-value workload characteristics published recently by Facebook [3].

First, *queries for small objects dominate*. Most keys are smaller than 32 bytes and most values no more than a few hundred bytes. In particular, there is one common type of request that almost exclusively uses 16 or 21 Byte keys and 2 Byte values.

The consequence of storing such small key-value objects is high memory overhead. Memcached always allocates a 56-Byte header (on 64-bit servers) for each key-value object *regardless of the size*. The header includes two pointers for the LRU linked list and one pointer for chaining to form the hash table. For small key-value objects, this space overhead cannot be amortized. Therefore we seek more memory efficient data structures for the index and cache.

Second, *queries are read heavy*. In general, a `GET`/`SET` ratio of 30:1 is reported for the Memcached workloads in Facebook. Important applications that can increase cache size on demand show even higher fractions of `GET`s (e.g., 99.8% are `GET`s, or `GET`/`SET`=500:1). Note that this ratio also depends on the GET hit ratio, because each GET miss is usually followed by a SET to update the cache by the application.

Though most queries are GETs, this operation is not optimized and locks are used extensively on the query path. For example, each GET operation must acquire (1) a lock for exclusive access to this particular key, (2) a global lock for exclusive access to the hash table; and (3) after reading the relevant key-value object, it must again acquire the global lock to update the LRU linked list. We aim to remove all mutexes on the GET path to boost the concurrency of Memcached.

3 Optimistic Concurrent Cuckoo Hashing

In this section, we present a compact, concurrent and cache-aware hashing scheme called *optimistic concurrent cuckoo hashing*. Compared with Memcached's original chaining-based hash table, our design improves memory efficiency by applying cuckoo hashing [23]—a practical, advanced hashing scheme with high memory efficiency and O(1) amortized insertion time and retrieval. However, basic cuckoo hashing does not support concurrent read/write access; it also requires multiple memory references for each insertion or lookup. To overcome these limitations, we propose a collection of new techniques that improve basic cuckoo hashing in concurrency, memory efficiency and cache-friendliness:

- An *optimistic version* of cuckoo hashing that supports multiple-reader/single writer concurrent access, while preserving its space benefits;
- A technique using a short summary of each key to improve the cache locality of hash table operations; and
- An optimization for cuckoo hashing insertion that improves throughput.

As we show in Section 5, combining these techniques creates a hashing scheme that is attractive in practice: its hash table achieves over 90% occupancy (compared to 50% for linear probing, or needing the extra pointers required by chaining) [**?**]. Each lookup requires only two *parallel* cacheline reads followed by (up to) one memory reference on average. In contrast, naive cuckoo hashing requires two parallel cacheline reads followed by (up to) 2*N* parallel memory references if each bucket has *N* keys; and chaining requires (up to) *N dependent* memory references to scan a bucket of *N* keys. The hash table supports multiple readers and a single writer, substantially speeding up read-intensive workloads while maintaining equivalent performance for write-heavy workloads.

Interface The hash table provides `Lookup`, `Insert` and `Delete` operations for indexing all key-value objects. On `Lookup`, the hash table returns a pointer to the relevant key-value object, or "does not exist" if the key can not be found. On `Insert`, the hash table returns *true* on success, and *false* to indicate the hash table is too full.[2] `Delete` simply removes the key's entry from the hash table. We focus on `Lookup` and `Insert` as `Delete` is very similar to `Lookup`.

Basic Cuckoo Hashing Before presenting our techniques in detail, we first briefly describe how to perform cuckoo hashing. The basic idea of cuckoo hashing is to use two hash functions instead of one, thus providing each key two possible locations where it can reside. Cuckoo hashing can dynamically relocate existing keys and refine the table to make room for new keys during insertion.

Our hash table, as shown in Figure 2, consists of an array of *buckets*, each having 4 *slots*.[3] Each slot contains a *pointer* to the key-value object and a short summary of

[2] As in other hash table designs, an expansion process can increase the cuckoo hash table size to allow for additional inserts.

[3] Our hash table is 4-way set-associative. Without set-associativity, basic cuckoo hashing allows only 50% of the table entries to be occupied before unresolvable collisions occur. It is possible to improve the space utilization to over 90% by using a 4-way (or higher) set associative hash table. [9]

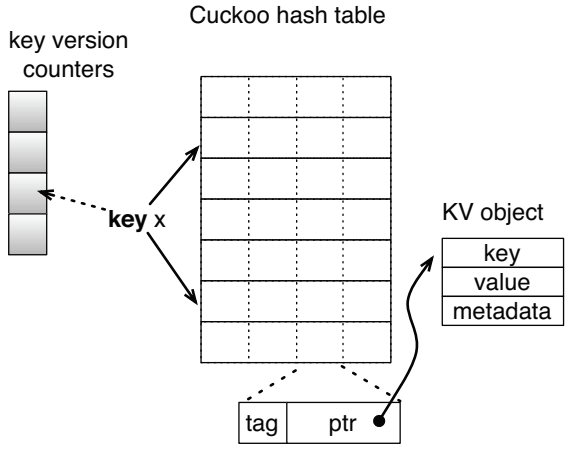


Figure 2: Hash table overview: The hash table is 4-way set-associative. Each key is mapped to 2 buckets by hash functions and associated with 1 version counter; Each slot stores a tag of the key and a pointer to the key-value item. Values in gray are used for optimistic locking and must be accessed atomically.

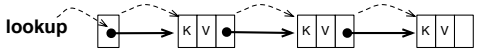
the key called a *tag*. To support keys of variable length, the full keys and values are not stored in the hash table, but stored with the associated metadata outside the table and referenced by the pointer. A null pointer indicates this slot is not used.

Each key is mapped to two random buckets, so `Lookup` checks all 8 candidate keys from every slot. To insert a new key x into the table, if either of the two buckets has an empty slot, it is then inserted in that bucket; if neither bucket has space, `Insert` selects a random key y from one candidate bucket and relocates y to its own alternate location. Displacing y may also require kicking out another existing key z, so this procedure may repeat until a vacant slot is found, or until a maximum number of displacements is reached (e.g., 500 times in our implementation). If no vacant slot found, the hash table is considered too full to insert and an expansion process is scheduled. Though it may execute a sequence of displacements, the amortized insertion time of cuckoo hashing is $O(1)$ [23].

3.1 Tag-based Lookup/Insert

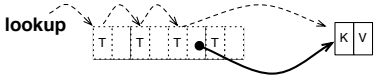
To support keys of variable length and keep the index compact, the actual keys are not stored in the hash table and must be retrieved by following a pointer. We propose a cache-aware technique to perform cuckoo hashing with minimum memory references by using *tags*—a short hash of the keys (one-byte in our implementation). This technique is inspired by "partial-key cuckoo hashing" which we proposed in previous work [17], but eliminates the prior approach's limitation in the maximum table size.

Cache-friendly Lookup The original Memcached lookup is not cache-friendly. It requires multiple *dependent* pointer dereferences to traverse a linked list:



Neither is basic cuckoo hashing cache-friendly: checking two buckets on each `Lookup` makes up to 8 (parallel) pointer dereferences. In addition, displacing each key on `Insert` also requires a pointer dereference to calculate the alternate location to swap, and each `Insert` may perform several displacement operations.

Our hash table eliminates the need for pointer dereferences in the common case. We compute a 1-Byte tag as the summary of each inserted key, and store the tag in the same bucket as its pointer. `Lookup` first compares the tag, then retrieves the full key only if the tag matches. This procedure is as shown below (T represents the tag)



It is possible to have false retrievals due to two different keys having the same tag, so the fetched full key is further verified to ensure it was indeed the correct one. With a 1-Byte tag by hashing, the chance of tag-collision is only $1/2^8 = 0.39\%$. After checking all 8 candidate slots, a negative `Lookup` makes $8 \times 0.39\% = 0.03$ pointer dereferences on average. Because each bucket fits in a CPU cacheline (usually 64-Byte), on average each `Lookup` makes only 2 parallel cacheline-sized reads for checking the two buckets plus either 0.03 pointer dereferences if the `Lookup` misses or 1.03 if it hits.

Cache-friendly Insert We also use the tags to avoid retrieving full keys on `Insert`, which were originally needed to derive the alternate location to displace keys. To this end, our hashing scheme computes the two candidate buckets b_1 and b_2 for key x by

$$\begin{aligned} b_1 &= \text{HASH}(x) && \text{// based on the entire key} \\ b_2 &= b_1 \oplus \text{HASH(tag)} && \text{// based on } b_1 \text{ and tag of } x \end{aligned}$$

b_2 is still a random variable uniformly distributed[4]; more importantly b_1 can be computed by the same formula from b_2 and tag. This property ensures that to displace a key originally in bucket b—no matter if b is b_1 or b_2— it is possible to calculate its alternate bucket b' from bucket index b and the tag stored in bucket b by

$$b' = b \oplus \text{HASH(tag)} \tag{1}$$

As a result, `Insert` operations can operate using only information in the table and never have to retrieve keys.

[4] b_2 is no longer fully independent from b_1. For a 1-Byte tag, there are up to 256 different values of b_2 given a specific b_1. Microbenchmarks in Section 5 show that our algorithm still achieves close-to-optimal load factor, even if b_2 has some dependence on b_1.

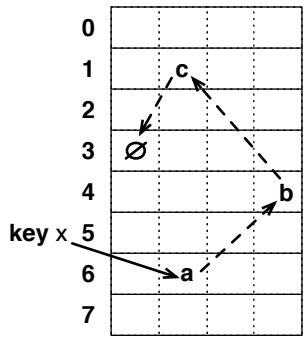


Figure 3: Cuckoo path. ∅ represents an empty slot.

3.2 Concurrent Cuckoo Hashing

Effectively supporting concurrent access to a cuckoo hash table is challenging. A previously proposed scheme improved concurrency by trading space [12]. Our hashing scheme is, to our knowledge, the first approach to support concurrent access (multi-reader/single-writer) while still maintaining the high space efficiency of cuckoo hashing (e.g., > 90% occupancy).

For clarity of presentation, we first define a *cuckoo path* as the sequence of displaced keys in an `Insert` operation. In Figure 3 "$a \Rightarrow b \Rightarrow c$" is one cuckoo path to make one bucket available to insert key x.

There are two major obstacles to making the sequential cuckoo hashing algorithm concurrent:

1. *Deadlock risk (writer/writer):* An `Insert` may modify a set of buckets when moving the keys along the cuckoo path until one key lands in an available bucket. It is not known *before swapping the keys* how many and which buckets will be modified, because each displaced key depends on the one previously kicked out. Standard techniques to make `Insert` atomic and avoid deadlock, such as acquiring all necessary locks in advance, are therefore not obviously applicable.
2. *False misses (reader/writer):* After a key is kicked out of its original bucket but before it is inserted to its alternate location, this key is unreachable from both buckets and temporarily unavailable. If `Insert` is not atomic, a reader may complete a `Lookup` and return a false miss during a key's unavailable time. E.g., in Figure 3, after replacing b with a at bucket 4, but before b relocates to bucket 1, b appears at neither bucket in the table. A reader looking up b at this moment may return negative results.

The only scheme previously proposed for concurrent cuckoo hashing [12] that we know of breaks up `Insert`s into a sequence of atomic displacements rather than locking the entire cuckoo path. It adds extra space at each bucket as an overflow buffer to temporarily host keys swapped from other buckets, and thus avoid kicking out existing keys. Hence, its space overhead (typically two more slots per bucket as buffer) is much higher than the basic cuckoo hashing.

Our scheme instead maintains high memory efficiency and also allows multiple-reader concurrent access to the hash table. To avoid writer/writer deadlocks, it allows only one writer at a time—a tradeoff we accept as our target workloads are read-heavy. To eliminate false misses, our design changes the order of the basic cuckoo hashing insertion by:

1) *separating discovering a valid cuckoo path from the execution of this path*. We first search for a cuckoo path, but do not move keys during this search phase.
2) *moving keys backwards along the cuckoo path*. After a valid cuckoo path is known, we first move the last key on the cuckoo path to the free slot, and then move the second to last key to the empty slot left by the previous one, and so on. As a result, each swap affects only one key at a time, which can always be successfully moved to its new location without any kickout.

Intuitively, the original `Insert` always moves a selected key to its other bucket and kicks out another existing key unless an empty slot is found in that bucket. Hence, there is always a victim key "floating" before `Insert` completes, causing false misses. In contrast, our scheme first discovers a cuckoo path to an empty slot, then propagates this empty slot towards the key for insertion along the path. To illustrate our scheme in Figure 3, we first find a valid cuckoo path "$a \Rightarrow b \Rightarrow c$" for key x without editing any buckets. After the path is known, c is swapped to the empty slot in bucket 3, followed by relocating b to the original slot of c in bucket 1 and so on. Finally, the original slot of a will be available and x can be directly inserted into that slot.

3.2.1 Optimization: Optimistic Locks for Lookup

Many locking schemes can work with our proposed concurrent cuckoo hashing, as long as they ensure that during `Insert`, all displacements along the cuckoo path are atomic with respect to `Lookup`s. The most straightforward scheme is to lock the two relevant buckets before each displacement and each `Lookup`. Though simple, this scheme requires locking twice for every `Lookup` and in a careful order to avoid deadlock.

Optimizing for the common case, our approach takes advantage of having a single writer to synchronize `Insert` and `Lookup`s with low overhead. Instead of locking on buckets, it assigns a version counter for each key, updates its version when displacing this key on `Insert`, and looks for a version change during `Lookup` to detect any concurrent displacement.

Lock Striping [12] The simplest way to maintain each key's version is to store it inside each key-value object. This approach, however, adds one counter for each key and there could be hundred of millions of keys. More importantly, this approach leads to a race condition: to check or update the version of a given key, we must first lookup in the hash table to find the key-value object (stored external to the hash table), and this initial lookup is not protected by any lock and thus not thread-safe.

Instead, we create an array of counters (Figure 2). To keep this array small, each counter is shared among multiple keys by hashing (e.g., the *i*-th counter is shared by all keys whose hash value is *i*). Our implementation keeps 8192 counters in total (or 32 KB). This permits the counters to fit in cache, but allows substantial concurrent access. It also keeps the chance of a "false retry" (re-reading a key due to modification of an unrelated key) to roughly 0.01%. All counters are initialized to 0 and only read/updated by atomic memory operations to ensure the consistency among all threads.

Optimistic Locking [15] Before displacing a key, an `Insert` process first increases the relevant counter by one, indicating to the other `Lookup`s an on-going update for this key; after the key is moved to its new location, the counter is again increased by one to indicate the completion. As a result, the key version is increased by 2 after each displacement.

Before a `Lookup` process reads the two buckets for a given key, it first snapshots the version stored in its counter: If this version is odd, there must be a concurrent `Insert` working on the same key (or another key sharing the same counter), and it should wait and retry; otherwise it proceeds to the two buckets. After it finishes reading both buckets, it snapshots the counter again and compares its new version with the old version. If two versions differ, the writer must have modified this key, and the `Lookup` should retry. The proof of correctness in the Appendix covers the corner cases.

3.2.2 Optimization: Multiple Cuckoo Paths

Our revised `Insert` process first looks for a valid cuckoo path before swapping the key along the path. Due to the separation of search and execution phases, we apply the following optimization to speed path discovery and increase the chance of finding an empty slot.

Instead of searching for an empty slot along one cuckoo path, our `Insert` process keeps track of multiple paths in parallel. At each step, multiple victim keys are "kicked out," each key extending its own cuckoo path. Whenever one path reaches an available bucket, this search phase completes.

With multiple paths to search, insert may find an empty slot earlier and thus improve the throughput. In addition, it improves the chance for the hash table to store a new key before exceeding the maximum number of displacements performed, thus increasing the load factor. The effect of having more cuckoo paths is evaluated in Section 5.

4 Concurrent Cache Management

Cache management and eviction is the second important component of MemC3. When serving small key-value objects, this too becomes a major source of space overhead in Memcached, which requires *18 Bytes for each key* (i.e., two pointers and a 2-Byte reference counter) to ensure that keys can be evicted safely in a strict LRU order. String LRU cache management is also a synchronization bottleneck, as all updates to the cache must be serialized in Memcached.

This section presents our efforts to make the cache management *space efficient* (1 bit per key) and *concurrent* (no synchronization to update LRU) by implementing an *approximate LRU cache* based on the CLOCK replacement algorithm [6]. CLOCK is a well-known algorithm; our contribution lies in integrating CLOCK replacement with the optimistic, striped locking in our cuckoo algorithm to reduce both locking and space overhead.

As our target workloads are dominated by small objects, the space saved by trading perfect for approximate LRU allows the cache to store siginifcantly more entries, which in turn improves the hit ratio. As we will show in Section 5, our cache management achieves 3× to 10× the query throughput of the default cache in Memcached, while also improving the hit ratio.

CLOCK Replacement A cache must implement two functions related to its replacement policy:

- `Update` to keep track of the recency after querying a key in the cache; and
- `Evict` to select keys to purge when inserting keys into a full cache.

Memcached keeps each key-value entry in a doubly-linked-list based LRU queue within its own slab class. After each cache query, `Update` moves the accessed entry to the *head* of its own queue; to free space when the cache is full, `Evict` replaces the entry on the *tail* of the queue by the new key-value pair. This ensures strict LRU eviction in each queue, but unfortunately it also requires two pointers per key for the doubly-linked list and, more importantly, all `Updates` to one linked list are serialized. Every read access requires an update, and thus the queue permits no concurrency even for read-only workloads.

CLOCK approximates LRU with improved concurrency and space efficiency. For each slab class, we maintain a *circular buffer* and a *virtual hand*; each bit in the

buffer represents the recency of a different key-value object: 1 for "recently used" and 0 otherwise. Each `Update` simply sets the recency bit to 1 on each key access; each `Evict` checks the bit currently pointed by the hand. If the current bit is 0, `Evict` selects the corresponding key-value object; otherwise we reset this bit to 0 and advance the hand in the circular buffer until we see a bit of 0.

Integration with Optimistic Cuckoo Hashing The `Evict` process must coordinate with reader threads to ensure the eviction is safe. Otherwise, a key-value entry may be overwritten by a new (key,value) pair after eviction, but threads still accessing the entry for the evicted key may read dirty data. To this end, the original Memcached adds to each entry a 2-Byte reference counter to avoid this rare case. Reading this per-entry counter, the `Evict` process knows how many other threads are accessing this entry concurrently and avoids evicting those busy entries.

Our cache integrates cache eviction with our optimistic locking scheme for cuckoo hashing. When `Evict` selects a victim key x by CLOCK, it first increases key x's version counter to inform other threads currently reading x to retry; it then deletes x from the hash table to make x unreachable for later readers, including those retries; and finally it increases key x's version counter again to complete the change for x. Note that `Evict` and the hash table `Insert` are both serialized (using locks) so when updating the counters they can not affect each other.

With `Evict` as above, our cache ensures consistent `GET`s by version checking. Each `GET` first snapshots the version of the key before accessing the hash table; if the hash table returns a valid pointer, it follows the pointer and reads the value assoicated. Afterwards, `GET` compares the latest key version with the snapshot. If the verions differ, then `GET` may have observed an inconsistent intermediate state and must retry. The pseudo-code of `GET` and `SET` is shown in Algorithm 1.

Algorithm 1: Psuedo code of `SET` and `GET`

```
SET(key, value)   //insert (key,value) to cache
begin
    lock();
    ptr = Alloc();      //try to allocate space
    if ptr == NULL then
        ptr = Evict();  //cache is full, evict old item
    memcpy key, value to ptr;
    Insert(key, ptr); //index this key in hashtable
    unlock();

GET(key)   //get value of key from cache
begin
    while true do
        vs = ReadCounter(key);   //key version
        ptr= Lookup(key);        //check hash table
        if ptr == NULL then return NULL ;
        prepare response for data in ptr;
        ve = ReadCounter(key);   //key version
        if vs & 1 or vs != ve then
            //may read dirty data, try again
            continue
        Update(key);             //update CLOCK
        return response
```

5 Evaluation

This section investigates how the proposed techniques and optimizations contribute to performance and space efficiency. We "zoom out" the evaluation targets, starting with the hash table itself, moving to the cache (including the hash table and cache eviction management), and concluding with the full MemC3 system (including the cache and network). With all optimizations combined, MemC3 achieves 3× the throughput of Memcached. Our proposed core hash table if isolated can achieve 5 million lookups/sec per thread and 35 million lookups/sec when accessed by 12 threads.

5.1 Platform

All experiments run on a machine with the following configuration. The CPU of this server is optimized for energy efficiency rather than high performance, and our system is CPU intensive, so we expect the absolute performance would be higher on "beefier" servers.

CPU	2× Intel Xeon L5640 @ 2.27GHz
# cores	2×6
LLC	2×12 MB L3-cache
DRAM	2×16 GB DDR SDRAM
NIC	10Gb Ethernet

5.2 Hash Table Microbenchmark

In the following experiments, we first benchmark the construction of hash tables and measure the space efficiency. Then we examine the lookup performance of a single thread and the aggregate throughput of 6 threads all accessing the same hash table, to analyze the contribution of different optimizations. In this subsection, hash tables are linked into a workload generator directly and benchmarked on a local machine.

Space Efficiency and Construction Speed We insert unique keys into empty cuckoo and chaining hash tables using a single thread, until each hash table reaches its maximum capacity. The chaining hash table, as used in Mem-

Hash table	Size (MB)	# keys (million)	Byte/key	Load factor	Construction rate (million keys/sec)	Largest bucket
Chaining	1280	100.66	13.33	–	14.38	13
Cuckoo 1path	1152	127.23	9.49	94.79%	6.19	4
Cuckoo 2path	1152	127.41	9.48	94.93%	7.43	4
Cuckoo 3path	1152	127.67	9.46	95.20%	7.29	4

Table 2: Comparison of space efficiency and construction speed of hash tables. Results in this table are independent of the key-value size. Each data point is the average of 10 runs.

cached, stops insertion if $1.5n$ objects are inserted to a table of n buckets to prevent imbalanced load across buckets; our cuckoo hash table stops when a single `Insert` fails to find an empty slot after 500 consecutive displacements. We initialize both types of hash tables to have a similar size (around 1.2 GB, including the space cost for pointers)

Table 2 shows that the cuckoo hash table is much more compact. Chaining requires 1280 MB to index 100.66 million items (i.e., 13.33 bytes per key); cuckoo hash tables are both smaller in size (1152 MB) and contain at least 20% more items, using no more than 10 bytes to index each key. Both cuckoo and chaining hash tables store only pointers to objects rather than the real key-value data; the *index* size is reduced by 1/3. A smaller index matters more for small key-value pairs.

Table 2 also compares cuckoo hash tables using different numbers of cuckoo paths to search for empty slots (Section 3.2.2). All of the cuckoo hash tables have high occupancy (roughly 95%). While more cuckoo paths only slightly improve the load factor, they boost construction speed non-trivially. The table with 2-way search achieves the highest construction rate (7.43 MOPS), as searching on two cuckoo paths balances the chance to find an empty slot vs. the resources required to keep track of all paths.

Chaining table construction is twice as fast as cuckoo hashing, because each insertion requires modifying only the head of the chain. Though fast, its most loaded bucket contains 13 objects in a chain (the average bucket has 1.5 objects). In contrast, bucket size in a cuckoo hash table is fixed (i.e., 4 slots), making it a better match for our targeted read-intensive workloads.

Cuckoo Insert Although the amortized cost to insert one key with cuckoo hashing is O(1), it requires more displacements to find an empty slot when the table is more occupied. We therefore measure the insertion cost—in terms of both the number of displacements per insert and the latency—to a hash table with x% of all slots filled, and vary x from 0% to the maximum possible load factor. Using two cuckoo paths improves insertion latency, but using more than that has diminishing or negative returns. Figure 4 further shows the reciprocal throughput, expressed as latency. When the table is 70% filled, a cuckoo insert can complete within 100 ns. At 95% occupancy, insert delay is 1.3 μs with a single cuckoo path, and 0.84 μs using two.

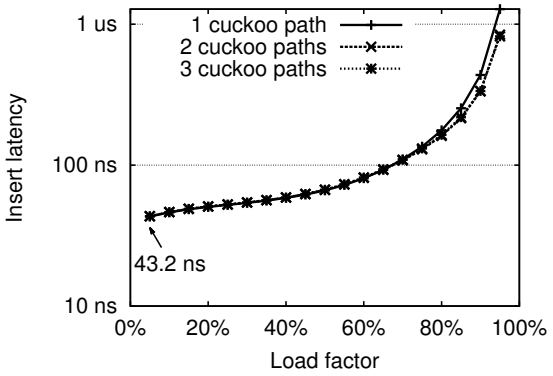
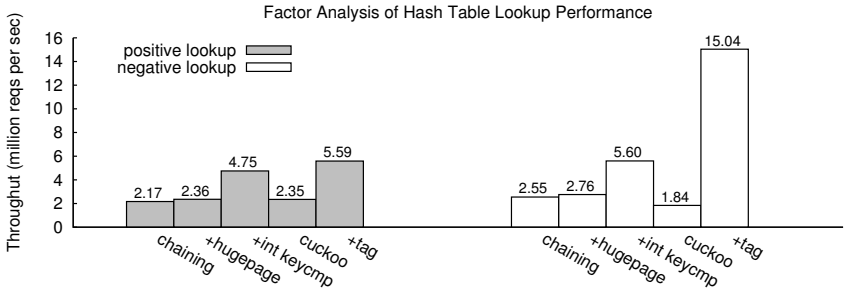


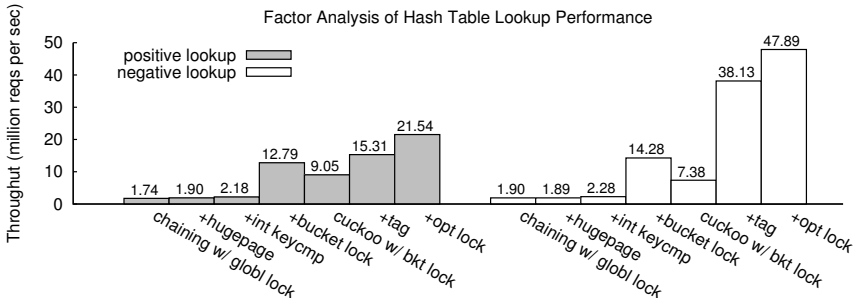
Figure 4: Cuckoo insert, with different number of parallel searches. Each data point is the average of 10 runs.

Factor Analysis of Lookup Performance This experiment investigates how much each optimization in Section 3 contributes to the hash table. We break down the performance gap between the basic chaining hash table used by Memcached and the final optimistic cuckoo hash table we proposed, and measure a set of hash tables—starting from the basic chaining and adding optimizations cumulatively as follows:

- **Chaining** is the default hash table of Memcached, serving as the baseline. A global lock is used to synchronize multiple threads.
- **+hugepage** enables 2MB x86 hugepage support in Linux to reduce TLB misses.
- **+int keycmp** replaces the default `memcmp` (used for full key comparison) by casting each key into an integer array and then comparing based on the integers.
- **+bucket lock** replaces the global lock by bucket-based locks.
- **cuckoo** applies the naive cuckoo hashing to replace chaining, without storing the tags in buckets and using bucket-based locking to coordinate multiple threads.



(a) Single-thread lookup performance with non-concurrency optimizations, all locks disabled



(b) Aggregate lookup performance of 6 threads with all optimizations

Figure 5: Contribution of optimizations to the hash table lookup performance. Optimizations are cumulative. Each data point is the average of 10 runs.

- **+tag** stores the 1-Byte hash for each key to improve cache-locality for both `Insert` and `Lookup` (Section 3.1).
- **+opt lock** replaces the per-bucket locking scheme by optimistic locking to ensure atomic displacement (Section 3.2.1).

Single-thread lookup performance is shown in Figure 5a with lookups all positive or all negative. No lock is used for this experiment. In general, combining all optimizations improves performance by ~ 2× compared to the naive chaining in Memcached for positive lookups, and by ~ 5× for negative lookups. Enabling "hugepage" improves the baseline performance slightly; while "int keycmp" can almost double the performance over "hugepage" for both workloads. This is because our keys are relatively small, so the startup overhead in the built-in `memcmp` becomes relatively large. Using cuckoo hashing without the "tag" optimization *reduces* performance, because naive cuckoo hashing requires more memory references to retrieve the keys in all $4 \times 2 = 8$ candidate locations on each lookup (as described in Section 3.1). The "tag" optimization therefore significantly improves the throughput of read-only workloads (2× for positive lookups and 8× for negative lookups), because it compares the 1-byte tag first before fetching the real keys outside the table and thus eliminates a large fraction of CPU cache misses.

Multi-thread lookup performance is shown in Figure 5b, measured by aggregating the throughput from 6 threads accessing the same hash table. Different from the previous experiment, a global lock is used for the baseline chaining (as in Memcached by default) and replaced by per-bucket locking and finally optimistic locking for the cuckoo hash table.

The performance gain (~ 12× for positive and ~ 25× for negative lookups) of our proposed hashing scheme over the default Memcached hash table is large. In Memcached, all hash table operations are serialized by a global lock, thus the basic chaining hash table in fact performs worse than its single-thread throughput in Figure 5a. The slight improvement (< 40%) from "hugepage" and "int keycmp" indicates that most performance benefit is from making the data structures concurrent. The "bucket lock" optimization replaces the global lock in chaining hash tables and thus significantly improves the performance by 5× to 6× compared to "int keycmp". Using the basic concurrent cuckoo reduces throughput (due to unnecessary memory references), while the "tag" optimization is again essential to boost the performance of cuckoo hashing and outperform chaining with per-bucket locks. Finally, the optimistic locking scheme further improves the performance significantly.

Multi-core Scalability Figure 6 illustrates how the total hash table throughput changes as more threads access

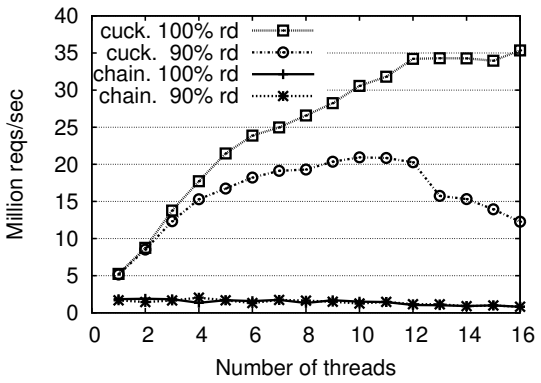


Figure 6: Hash table throughput vs. number of threads. Each data point is the average of 10 runs.

the same hash table. We evaluate read-only and 10% write workloads. The throughput of the default hash table does not scale for either workload, because all hash table operations are serialized. Due to lock contention, the throughput is actually lower than the single-thread throughput without locks.

Using our proposed cuckoo hashing for the read-only workload, the performance scales linearly to 6 threads because each thread is pinned on a dedicated physical core on the same 6-core CPU. The next 6 threads are pinned to the other 6-core CPU in the same way. The slope of the curve becomes lower due to cross-CPU memory traffic. Threads after the first 12 are assigned to already-busy cores, and thus performance does not further increase.

With 10% `Insert`, our cuckoo hashing reaches a peak performance of 20 MOPS at 10 threads. Each `Insert` requires a lock to be serialized, and after 10 threads the lock contention becomes the bottleneck.

We further vary the fraction of insert queries in the workload and measure the best performance achieved by different hash tables. Figure 7 shows this best performance and also the number of threads (between 1 and 16) required to achieve this performance. In general, cuckoo hash tables outperform chaining hash tables. When more write traffic is generated, performance of cuckoo hash tables declines because `Inserts` are serialized and more `Lookups` happen concurrently. Consequently, the best performance for 10% insert is achieved using only 9 threads; while with 100% lookup, it scales to 16 threads. Whereas the best performance of chaining hash tables (with either a global lock or per-bucket locks) keeps roughly the same when the workloads become more write-intensive.

5.3 Cache Microbenchmark

Workload We use YCSB [5] to generate 100 million key-value queries, following a zipf distribution. Each key is 16 Bytes and each value 32 Bytes. We evaluate caches with four configurations:

- **chaining+LRU**: the default Memcached cache configuration, using chaining hash table to index keys and LRU for replacement;
- **cuckoo+LRU**: keeping LRU, but replacing the hash table by concurrent optimistic cuckoo hashing with all optimizations proposed;
- **chaining+CLOCK**: an alternative baseline combining optimized chaining with the CLOCK replacement algorithm. Because CLOCK requires no serialization to update, we also replace the global locking in the chaining hash table with the per-bucket locks; we further include our engineering optimizations such as "hugepage", "int keycmp".
- **cuckoo+CLOCK**: the data structure of MemC3, using cuckoo hashing to index keys and CLOCK for replacement.

We vary the cache size from 64 MB to 10 GB. Note that this cache size parameter does not count the space for the hash table, only the space used to store key-value objects. All four types of caches are linked into a workload generator and micro-benchmarked locally.

Cache Throughput Because each `GET` miss is followed by a `SET` to the cache, to understand the cache performance with heavier or lighter insertion load, we evaluate two settings:

- a read-only workload on a "big" cache (i.e., 10 GB, which is larger than the working set), which had no cache misses or inserts and is the best case for performance;
- a write-intensive workload on a "small" cache (i.e., 1 GB, which is ~10% of the total working set) where about 15% `GETs` miss the cache. Since each miss triggers a `SET` in turn, a workload with 15% inserts is worse than the typical real-world workload reported by Facebook [3].

Figure 8a shows the results of benchmarking the "big cache". Though there are no inserts, the throughput does not scale for the default cache (chaining+LRU), due to lock contention on each LRU update (moving an object to the head of the linked list). Replacing default chaining with the concurrent cuckoo hash table improves the peak throughput slightly. This suggests that only having a concurrent hash table is not enough for high performance. After replacing the global lock with bucket-based locks and removing the LRU synchronization bottleneck by using CLOCK, the chaining-based cache achieves 22

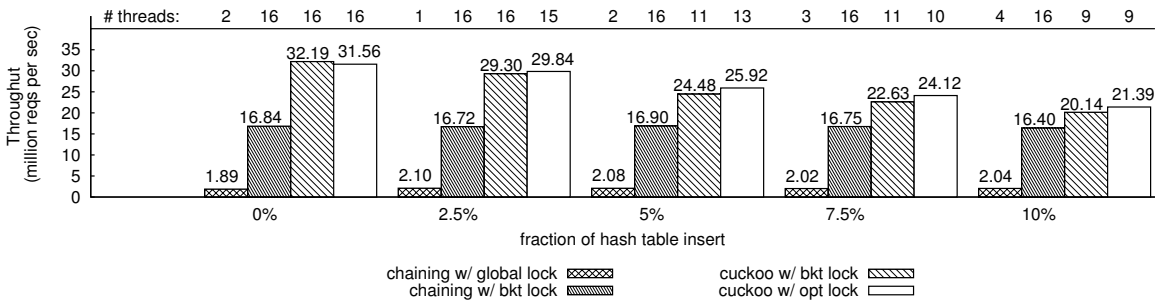
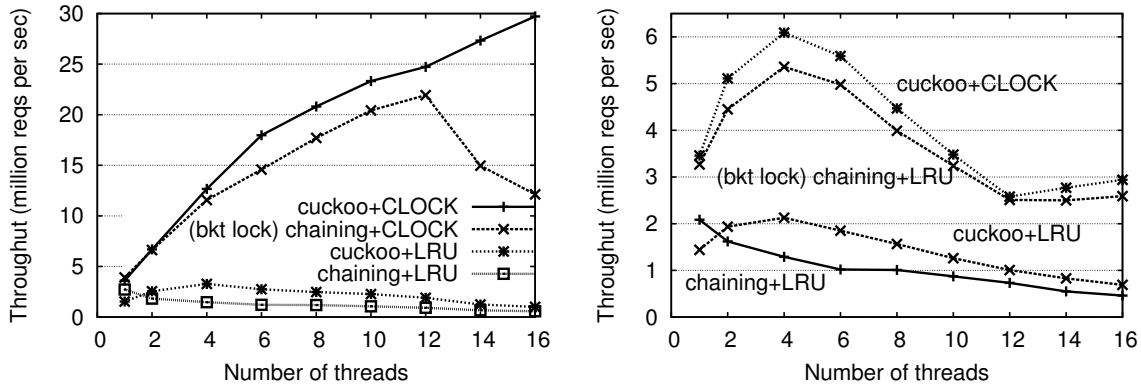


Figure 7: Tradeoff between lookup and insert performance. Best read + write throughput (with the number of threads needed to achieve it) is shown. Each data point is the average of 10 runs.



(a) 10GB "big cache" (> working set): 100% GETs hit

(b) 1GB "small cache" (< working set): 85% GETs hit

Figure 8: Cache throughput vs. number of threads. Each data point is the average of 10 runs.

MOPS at 12 threads, and drops quickly due to the CPU overhead for lock contention after all 12 physical cores are assigned. Our proposed cuckoo hash table combined with CLOCK, however, scales to 30 MOPS at 16 threads.

Figure 8b shows that peak performance is achieved at 6 MOPS for the "small cache" by combining CLOCK and cuckoo hashing. The throughput drop is because the 15% GET misses result in about 15% hash table inserts, so throughput drops after 6 threads due to serialized inserts.

Space Efficiency Table 3 compares the maximum number of items (16-Byte key and 32-Byte value) a cache can store given different cache sizes[5]. The default LRU with chaining is the least memory efficient scheme. Replacing chaining with cuckoo hashing improves the space utilization slightly (7%), because one pointer (for hash table chaining) is eliminated from each key-value object. Keeping chaining but replacing LRU with CLOCK improves space efficiency by 27% because two pointers (for LRU) and one reference count are saved per object. Combining CLOCK with cuckoo increases the space efficiency by 40% over the default. The space benefit arises from eliminating three pointers and one reference count per object.

Cache Miss Ratio Compared to the linked list based approach in Memcached, CLOCK approximates LRU eviction with much lower space overhead. This experiment sends 100 million queries (95% GET and 5% SET, in zipf distribution) to a cache with different configurations, and measures the resulting cache miss ratios. Note that each GET miss will trigger a retrieval to the backend database system, therefore reducing the cache miss ratio from 10% to 7% means a reduction of traffic to the backend by 30%. Table 3 shows when the cache size is smaller than 256 MB, the LRU-based cache provides a lower miss ratio than CLOCK. LRU with cuckoo hashing improves upon LRU with chaining, because it can store more items. In this experiment, 256 MB is only about 2.6% of the 10 GB working set. Therefore, when the cache size is very small, CLOCK—which is an

[5] The space to store the index hash tables is separate from the given cache space in Table 3. We set the hash table capacity larger than the maximum number of items that the cache space can possibly allocate. If chaining is used, the chaining pointers (inside each key-value object) are also allocated from the cache space.

	cache type	cache size					
		64 MB	128 MB	256 MB	512 MB	1 GB	2 GB
# items stored (million)	chaining+LRU	0.60	1.20	2.40	4.79	9.59	19.17
	cuckoo+LRU	0.65	1.29	2.58	5.16	10.32	20.65
	chaining+CLOCK	0.76	1.53	3.05	6.10	12.20	24.41
	cuckoo+CLOCK	**0.84**	**1.68**	**3.35**	**6.71**	**13.42**	**26.84**
cache miss ratio 95% GET, 5% SET zipf distribution	chaining+LRU	36.34%	31.91%	27.27%	22.30%	16.80%	10.44%
	cuckoo+LRU	**35.87%**	**31.42%**	**26.76%**	21.74%	16.16%	9.80%
	chaining+CLOCK	37.07%	32.51%	27.63%	22.20%	15.96%	8.54%
	cuckoo+CLOCK	36.46%	31.86%	26.92%	**21.38%**	**14.68%**	**7.89%**

Table 3: Comparison of four types of caches. Results in this table depend on the object size (16-Byte key and 32-Byte value used). Bold entries are the best in their columns. Each data point is the average of 10 runs.

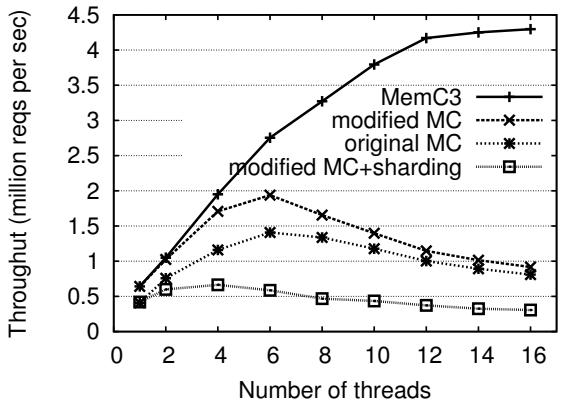


Figure 9: Full system throughput (over network) v.s. number of server threads

approximation—has a higher chance of evicting popular items than strict LRU. For larger caches, CLOCK with cuckoo hashing outperforms the other two schemes because the extra space improves the hit ratio more than the loss of precision decreases it.

5.4 Full System Performance

Workload This experiment uses the same workload as in Section 5.3, with 95% GETs and 5% SETs generated by YCSB with zipf distribution. MemC3 runs on the same server as before, but the clients are 50 different nodes connected by a 10GB Ethernet. The clients use libmemcached 1.0.7 [16] to communicate with our MemC3 server over the network. To amortize the network overhead, we use multi-get supported by libmemcached [16] by batching 100 GETs.

In this experiment, we compare four different systems: original Memcached, optimized Memcached (with non-algorithmic optimizations such as "hugepage", "in key-cmp" and tuned CPU affinity), optimized Memcached with sharding (one core per Memcached instance) and MemC3 with all optimizations enabled. Each system is allocated with 1GB memory space (not including hash table space).

Throughput Figure 9 shows the throughput as more server threads are used. Overall, the maximum throughput of MemC3 (4.4 MOPS) is almost 3× that of the original Memcached (1.5 MOPS). The non-algorithmic optimizations improve throughput, but their contribution is dwarfed by the algorithmic and data structure-based improvements.

A surprising result is that today's popular technique, *sharding, performs the worst* in this experiment. This occurs because the workload generated by YCSB is heavy-tailed, and therefore imposes differing load on the memcached instances. Those serving "hot" keys are heavily loaded while the others are comparatively idle. While the severity of this effect depends heavily upon the workload distribution, it highlights an important benefit of MemC3's approach of sharing all data between all threads.

6 Related Work

This section presents two categories of work most related to MemC3: efforts to improve individual key-value storage nodes in terms of throughput and space efficiency; and the related work applying cuckoo hashing.

Flash-based key-value stores such as BufferHash [1], FAWN-DS [2], SkimpyStash [8] and SILT [17] are optimized for I/O to external storage such as SSDs (e.g., by batching, or log-structuring small writes). Without slow I/O, the optimization goals for MemC3 are saving memory and eliminating synchronization. Previous work in *memory-based key-value stores* [4, 20, 13] boost performance on multi-core CPUs or GP-GPUs by sharding data to dedicated cores to avoid synchronization. MemC3 instead targets read-mostly workloads and deliberately avoids sharding to ensure high performance even for "hot" keys. Similar to MemC3, Masstree [18] also applied ex-

tensive optimizations for cache locality and optimistic concurrency control, but used very different techniques because it was a variation of B+-tree to support range queries. RAMCloud [22] focused on fast data reconstruction from on-disk replicas. In contrast, as a cache, MemC3 specifically takes advantage of the transience of the data it stores to improve space efficiency.

Cuckoo hashing [23] is an open-addressing hashing scheme with high space efficiency that assigns multiple candidate locations to each item and allows inserts to kick existing items to their candidate locations. FlashStore [7] applied cuckoo hashing by assigning each item 16 locations so that each lookup checks up to 16 locations, while our scheme requires reading only 2 locations in the hash table. We previously proposed *partial key cuckoo hashing* in the SILT system [17] to achieve high occupancy with only two hash functions, but our earlier algorithm limited the maximum hash table size and was therefore unsuitable for large in-memory caches. Our improved algorithm eliminates this limitation while retaining high memory efficiency. To make cuckoo operations concurrent, the prior approach of Herlihy et al. [12] traded space for concurrency. In contrast, our optimistic locking scheme allows concurrent readers without losing space efficiency.

7 Conclusion

MemC3 is an in-memory key-value store that is designed to provide caching for read-mostly workloads. It is built on carefully designed and engineered algorithms and data structures with a set of architecture-aware and workload-aware optimizations to achieve high concurrency, space-efficiency and cache-locality. In particular, MemC3 uses a new hashing scheme—optimistic cuckoo hashing—that achieves over 90% space occupancy and allows concurrent read access without locking. MemC3 also employs CLOCK-based cache management with only 1-bit per entry to approximate LRU eviction. Compared to Memcached, it reduces space overhead by more than 20 Bytes per entry. Our evaluation shows the throughput of our system is 3× higher than the original Memcached while storing 30% more objects for small key-value pairs.

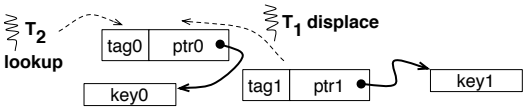
Acknowledgments

This work was supported by funding from the Intel Science and Technology Center for Cloud Computing, National Science Foundation award CCF-0964474, Google, and the PDL member companies. We thank the NSDI reviewers for their feedback, Iulian Moraru, Dong Zhou, Michael Mitzenmacher, and Rasmus Pagh for their valuable algorithmic suggestions, and Matthew Caesar for shepherding this paper.

A Correctness of the optimistic locking on keys

This appendix examines the possible interleavings of two threads in order to show that the optimistic locking scheme correctly prevents `Lookup` from returning wrong or corrupted data. Assume that threads T_1 and T_2 concurrently access the hash table. When both threads perform `Lookup`, correctness is trivial. When both `Insert`, they are serialized (`Insert` is guarded by a lock). The remaining case occurs when T_1 is `Insert` and T_2 is `Lookup`.

During `Insert`, T_1 may perform a sequence of displacement operations where each displacement is proceeded and followed by incrementing the counter. Without loss of generality, assume T_1 is displacing `key1` to a destination slot that originally hosts `key0`. Each slot contains a tag and a pointer, as shown:



`key0` differs from `key1` because there are no two identical keys in the hash table, which is guaranteed because every `Insert` effectively does a `Lookup` first. If T_2 reads the same slot as T_1 before T_1 completes its update:

case1: T_2 is looking for `key0` . Because `Insert` moves backwards along a cuckoo path (Section 3.2), `key0` must have been displaced to its other bucket (say bucket *i*), thus

- if T_2 has not checked bucket *i*, it will find `key0` when it proceeds to that bucket;
- if T_2 checked bucket *i* and did not find `key0` there, the operation that moves `key0` to bucket *i* must happen after T_2 reads bucket *i*. Therefore, T_2 will see a change in `key0`'s version counter and make a retry.

case2: T_2 is looking for `key1` . Since T_1 will atomically update `key1`'s version before and after the displacement, no matter what T_2 reads, it will detect the version change and retry.

case3: T_2 is looking for a key ≠ `key0` or `key1` . No matter what T_2 sees in the slot, it will be rejected eventually, either by the tags or by the full key comparison following the pointers. This is because the pointer field of this slot fetched by T_2 is either `ptr(key0)` or `ptr(key1)` rather than some corrupted pointer, ensured by the atomic read/write for 64-bit aligned pointers on 64-bit machines.[6] ∎

[6] quadword memory access aligned on a 64-bit boundary are atomic on Pentium and newer CPUs [14]

References

[1] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *Proc. 7th USENIX NSDI*, Apr. 2010.

[2] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. *Communications of the ACM*, 54(7): 101–109, July 2011.

[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 SIGMETRICS'12*, 2012.

[4] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *In Proceedings of the Second International Green Computing Conference*, Aug. 2011.

[5] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing (SOCC)*, June 2010.

[6] F. Corbato and M. I. O. T. C. P. MAC. *A Paging Experiment with the Multics System*. Defense Technical Information Center, 1968. URL `http://books.google.com/books?id=5wDQNwAACAAJ`.

[7] B. Debnath, S. Sengupta, and J. Li. FlashStore: high throughput persistent key-value store. *Proc. VLDB Endow.*, 3:1414–1425, Sept. 2010.

[8] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash. In *Proc. ACM SIGMOD*, June 2011.

[9] U. Erlingsson, M. Manasse, and F. Mcsherry. A cool and practical alternative to traditional hash tables. In *Seventh Whorkshop on Distributed Data and Structures (WDAS'2006)*, pages 1–6, 2006.

[10] Facebook Engineering Notes. Scaling memcached at Facebook. `http://www.facebook.com/note.php?note_id=39391378919`.

[11] N. Gunther, S. Subramanyam, and S. Parvu. Hidden scalability gotchas in memcached and friends. In *VELOCITY Web Performance and Operations Conference*, June 2010.

[12] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 0123705916, 9780123705914.

[13] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt. Characterizing and evaluating a key-value store application on heterogeneous cpu-gpu systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2012.

[14] intel-dev3a. Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3A. `http://www.intel.com/content/www/us/en/architecture-and-technology/`, 2011.

[15] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2): 213–226, June 1981. ISSN 0362-5915.

[16] libMemcached. libmemcached. `http://libmemcached.org/`, 2009.

[17] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[18] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. EuroSys*, pages 183–196, Apr. 2012.

[19] Memcached. A distributed memory object caching system. `http://memcached.org/`, 2011.

[20] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. Cphash: a cache-partitioned hash table. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2012.

[21] MongoDB. `http://mongodb.com`.

[22] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[23] R. Pagh and F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, May 2004.

[24] N. Provos. libevent. `http://monkey.org/~provos/libevent/`.

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

[26] VoltDB. VoltDB, the NewSQL database for high velocity applications. `http://voltdb.com/`.

Scaling Memcache at Facebook

Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li,
Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung,
Venkateshwaran Venkataramani

{rajeshn,hans}@fb.com, {sgrimm, marc}@facebook.com, {herman, hcli, rm, mpal, dpeek, ps, dstaff, ttung, veeve}@fb.com

Facebook Inc.

Abstract: `Memcached` is a well known, simple, in-memory caching solution. This paper describes how Facebook leverages `memcached` as a building block to construct and scale a distributed key-value store that supports the world's largest social network. Our system handles billions of requests per second and holds trillions of items to deliver a rich experience for over a billion users around the world.

1 Introduction

Popular and engaging social networking sites present significant infrastructure challenges. Hundreds of millions of people use these networks every day and impose computational, network, and I/O demands that traditional web architectures struggle to satisfy. A social network's infrastructure needs to (1) allow near real-time communication, (2) aggregate content on-the-fly from multiple sources, (3) be able to access and update very popular shared content, and (4) scale to process millions of user requests per second.

We describe how we improved the open source version of `memcached` [14] and used it as a building block to construct a distributed key-value store for the largest social network in the world. We discuss our journey scaling from a single cluster of servers to multiple geographically distributed clusters. To the best of our knowledge, this system is the largest `memcached` installation in the world, processing over a billion requests per second and storing trillions of items.

This paper is the latest in a series of works that have recognized the flexibility and utility of distributed key-value stores [1, 2, 5, 6, 12, 14, 34, 36]. This paper focuses on `memcached`—an open-source implementation of an in-memory hash table—as it provides low latency access to a shared storage pool at low cost. These qualities enable us to build data-intensive features that would otherwise be impractical. For example, a feature that issues hundreds of database queries per page request would likely never leave the prototype stage because it would be too slow and expensive. In our application, however, web pages routinely fetch thousands of key-value pairs from `memcached` servers.

One of our goals is to present the important themes that emerge at different scales of our deployment. While qualities like performance, efficiency, fault-tolerance, and consistency are important at all scales, our experience indicates that at specific sizes some qualities require more effort to achieve than others. For example, maintaining data consistency can be easier at small scales if replication is minimal compared to larger ones where replication is often necessary. Additionally, the importance of finding an optimal communication schedule increases as the number of servers increase and networking becomes the bottleneck.

This paper includes four main contributions: (1) We describe the evolution of Facebook's `memcached`-based architecture. (2) We identify enhancements to `memcached` that improve performance and increase memory efficiency. (3) We highlight mechanisms that improve our ability to operate our system at scale. (4) We characterize the production workloads imposed on our system.

2 Overview

The following properties greatly influence our design. First, users consume an order of magnitude more content than they create. This behavior results in a workload dominated by fetching data and suggests that caching can have significant advantages. Second, our read operations fetch data from a variety of sources such as MySQL databases, HDFS installations, and backend services. This heterogeneity requires a flexible caching strategy able to store data from disparate sources.

`Memcached` provides a simple set of operations (`set`, `get`, and `delete`) that makes it attractive as an elemental component in a large-scale distributed system. The open-source version we started with provides a single-machine in-memory hash table. In this paper, we discuss how we took this basic building block, made it more efficient, and used it to build a distributed key-value store that can process billions of requests per second. Hence-

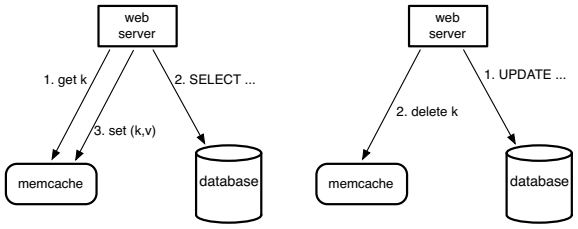


Figure 1: Memcache as a demand-filled look-aside cache. The left half illustrates the read path for a web server on a cache miss. The right half illustrates the write path.

forth, we use 'memcached' to refer to the source code or a running binary and 'memcache' to describe the distributed system.

Query cache: We rely on memcache to lighten the read load on our databases. In particular, we use memcache as a *demand-filled look-aside* cache as shown in Figure 1. When a web server needs data, it first requests the value from memcache by providing a string key. If the item addressed by that key is not cached, the web server retrieves the data from the database or other backend service and populates the cache with the key-value pair. For write requests, the web server issues SQL statements to the database and then sends a delete request to memcache that invalidates any stale data. We choose to delete cached data instead of updating it because deletes are idempotent. Memcache is not the authoritative source of the data and is therefore allowed to evict cached data.

While there are several ways to address excessive read traffic on MySQL databases, we chose to use memcache. It was the best choice given limited engineering resources and time. Additionally, separating our caching layer from our persistence layer allows us to adjust each layer independently as our workload changes.

Generic cache: We also leverage memcache as a more general key-value store. For example, engineers use memcache to store pre-computed results from sophisticated machine learning algorithms which can then be used by a variety of other applications. It takes little effort for new services to leverage the existing marcher infrastructure without the burden of tuning, optimizing, provisioning, and maintaining a large server fleet.

As is, memcached provides no server-to-server coordination; it is an in-memory hash table running on a single server. In the remainder of this paper we describe how we built a distributed key-value store based on memcached capable of operating under Facebook's workload. Our system provides a suite of configuration, aggregation, and routing services to organize memcached instances into a distributed system.

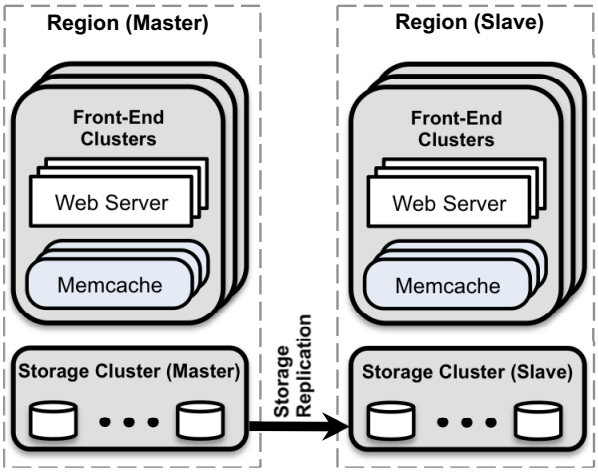


Figure 2: Overall architecture

We structure our paper to emphasize the themes that emerge at three different deployment scales. Our read-heavy workload and wide fan-out is the primary concern when we have one cluster of servers. As it becomes necessary to scale to multiple frontend clusters, we address data replication between these clusters. Finally, we describe mechanisms to provide a consistent user experience as we spread clusters around the world. Operational complexity and fault tolerance is important at all scales. We present salient data that supports our design decisions and refer the reader to work by Atikoglu *et al.* [8] for a more detailed analysis of our workload. At a high-level, Figure 2 illustrates this final architecture in which we organize co-located clusters into a region and designate a master region that provides a data stream to keep non-master regions up-to-date.

While evolving our system we prioritize two major design goals. (1) Any change must impact a user-facing or operational issue. Optimizations that have limited scope are rarely considered. (2) We treat the probability of reading transient stale data as a parameter to be tuned, similar to responsiveness. We are willing to expose slightly stale data in exchange for insulating a backend storage service from excessive load.

3 In a Cluster: Latency and Load

We now consider the challenges of scaling to thousands of servers within a cluster. At this scale, most of our efforts focus on reducing either the latency of fetching cached data or the load imposed due to a cache miss.

3.1 Reducing Latency

Whether a request for data results in a cache hit or miss, the latency of memcache's response is a critical factor in the response time of a user's request. A single user web request can often result in hundreds of individual

memcache get requests. For example, loading one of our popular pages results in an average of 521 distinct items fetched from memcache.[1]

We provision hundreds of memcached servers in a cluster to reduce load on databases and other services. Items are distributed across the memcached servers through consistent hashing [22]. Thus web servers have to routinely communicate with *many* memcached servers to satisfy a user request. As a result, *all* web servers communicate with every memcached server in a short period of time. This *all-to-all* communication pattern can cause incast congestion [30] or allow a single server to become the bottleneck for many web servers. Data replication often alleviates the single-server bottleneck but leads to significant memory inefficiencies in the common case.

We reduce latency mainly by focusing on the memcache client, which runs on each web server. This client serves a range of functions, including serialization, compression, request routing, error handling, and request batching. Clients maintain a map of all available servers, which is updated through an auxiliary configuration system.

Parallel requests and batching: We structure our web-application code to minimize the number of network round trips necessary to respond to page requests. We construct a directed acyclic graph (DAG) representing the dependencies between data. A web server uses this DAG to maximize the number of items that can be fetched concurrently. On average these batches consist of 24 keys per request[2].

Client-server communication: Memcached servers do not communicate with each other. When appropriate, we embed the complexity of the system into a stateless client rather than in the memcached servers. This greatly simplifies memcached and allows us to focus on making it highly performant for a more limited use case. Keeping the clients stateless enables rapid iteration in the software and simplifies our deployment process. Client logic is provided as two components: a library that can be embedded into applications or as a standalone proxy named mcrouter. This proxy presents a memcached server interface and routes the requests/replies to/from other servers.

Clients use UDP and TCP to communicate with memcached servers. We rely on UDP for get requests to reduce latency and overhead. Since UDP is connectionless, each thread in the web server is allowed to directly communicate with memcached servers directly, bypassing mcrouter, without establishing and maintaining a connection thereby reducing the overhead. The UDP implementation detects packets that are dropped or received out of order (using sequence numbers) and treats them as errors on the client side. It does not provide any mechanism to try to recover from them. In our infrastructure, we find this decision to be practical. Under peak load, memcache clients observe that 0.25% of get requests are discarded. About 80% of these drops are due to late or dropped packets, while the remainder are due to out of order delivery. Clients treat get errors as cache misses, but web servers will skip inserting entries into memcached after querying for data to avoid putting additional load on a possibly overloaded network or server.

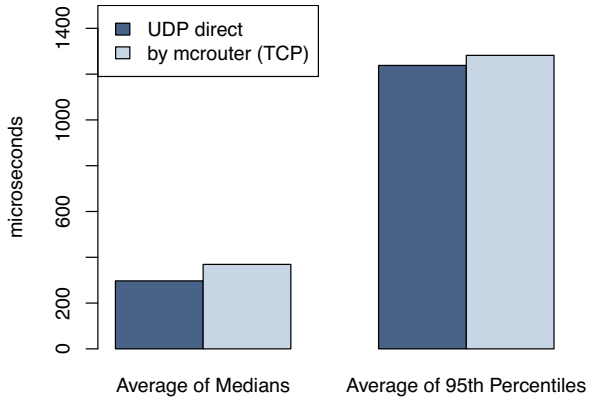


Figure 3: Get latency for UDP, TCP via mcrouter

For reliability, clients perform set and delete operations over TCP through an instance of mcrouter running on the same machine as the web server. For operations where we need to confirm a state change (updates and deletes) TCP alleviates the need to add a retry mechanism to our UDP implementation.

Web servers rely on a high degree of parallelism and over-subscription to achieve high throughput. The high memory demands of open TCP connections makes it prohibitively expensive to have an open connection between every web thread and memcached server without some form of connection coalescing via mcrouter. Coalescing these connections improves the efficiency of the server by reducing the network, CPU and memory resources needed by high throughput TCP connections. Figure 3 shows the average, median, and 95^{th} percentile latencies of web servers in production getting keys over UDP and through mcrouter via TCP. In all cases, the standard deviation from these averages was less than 1%. As the data show, relying on UDP can lead to a 20% reduction in latency to serve requests.

Incast congestion: Memcache clients implement flow-control mechanisms to limit incast congestion. When a

[1] The 95^{th} percentile of fetches for that page is 1,740 items.

[2] The 95^{th} percentile is 95 keys per request.

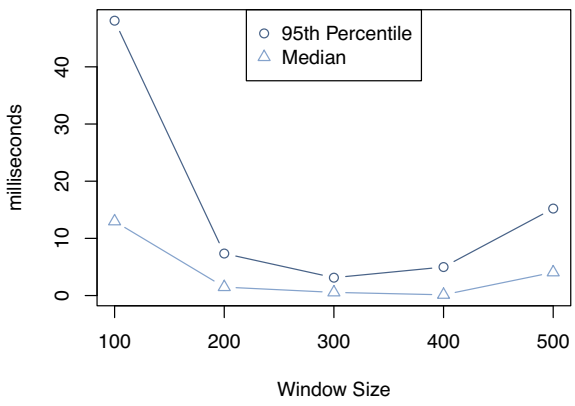


Figure 4: Average time web requests spend waiting to be scheduled

client requests a large number of keys, the responses can overwhelm components such as rack and cluster switches if those responses arrive all at once. Clients therefore use a sliding window mechanism [11] to control the number of outstanding requests. When the client receives a response, the next request can be sent. Similar to TCP's congestion control, the size of this sliding window grows slowly upon a successful request and shrinks when a request goes unanswered. The window applies to all `memcache` requests independently of destination; whereas TCP windows apply only to a single stream.

Figure 4 shows the impact of the window size on the amount of time user requests are in the runnable state but are waiting to be scheduled inside the web server. The data was gathered from multiple racks in one frontend cluster. User requests exhibit a Poisson arrival process at each web server. According to Little's Law [26], $L = \lambda W$, the number of requests queued in the server (L) is directly proportional to the average time a request takes to process (W), assuming that the input request rate is constant (which it was for our experiment). The time web requests are waiting to be scheduled is a direct indication of the number of web requests in the system. With lower window sizes, the application will have to dispatch more groups of `memcache` requests serially, increasing the duration of the web request. As the window size gets too large, the number of simultaneous `memcache` requests causes incast congestion. The result will be `memcache` errors and the application falling back to the persistent storage for the data, which will result in slower processing of web requests. There is a balance between these extremes where unnecessary latency can be avoided and incast congestion can be minimized.

3.2 Reducing Load

We use `memcache` to reduce the frequency of fetching data along more expensive paths such as database queries. Web servers fall back to these paths when the desired data is not cached. The following subsections describe three techniques for decreasing load.

3.2.1 Leases

We introduce a new mechanism we call *leases* to address two problems: stale sets and thundering herds. A stale set occurs when a web server sets a value in `memcache` that does not reflect the latest value that should be cached. This can occur when concurrent updates to `memcache` get reordered. A thundering herd happens when a specific key undergoes heavy read and write activity. As the write activity repeatedly invalidates the recently set values, many reads default to the more costly path. Our lease mechanism solves both problems.

Intuitively, a `memcached` instance gives a *lease* to a client to set data back into the cache when that client experiences a cache miss. The lease is a 64-bit token bound to the specific key the client originally requested. The client provides the lease token when setting the value in the cache. With the lease token, `memcached` can verify and determine whether the data should be stored and thus arbitrate concurrent writes. Verification can fail if `memcached` has invalidated the lease token due to receiving a delete request for that item. Leases prevent stale sets in a manner similar to how load-link/store-conditional operates [20].

A slight modification to *leases* also mitigates thundering herds. Each `memcached` server regulates the rate at which it returns tokens. By default, we configure these servers to return a token only once every 10 seconds per key. Requests for a key's value within 10 seconds of a token being issued results in a special notification telling the client to wait a short amount of time. Typically, the client with the lease will have successfully set the data within a few milliseconds. Thus, when waiting clients retry the request, the data is often present in cache.

To illustrate this point we collect data for all cache misses of a set of keys particularly susceptible to thundering herds for one week. Without leases, all of the cache misses resulted in a peak database query rate of 17K/s. With leases, the peak database query rate was 1.3K/s. Since we provision our databases based on peak load, our lease mechanism translates to a significant efficiency gain.

Stale values: With leases, we can minimize the application's wait time in certain use cases. We can further reduce this time by identifying situations in which returning slightly out-of-date data is acceptable. When a key is deleted, its value is transferred to a data struc-

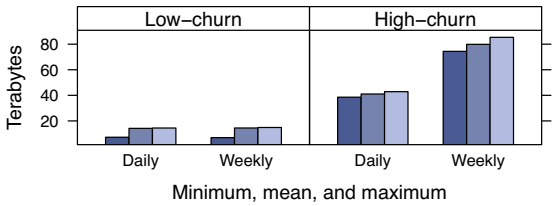


Figure 5: Daily and weekly working set of a high-churn family and a low-churn key family

ture that holds recently deleted items, where it lives for a short time before being flushed. A get request can return a lease token or data that is marked as stale. Applications that can continue to make forward progress with stale data do not need to wait for the latest value to be fetched from the databases. Our experience has shown that since the cached value tends to be a monotonically increasing snapshot of the database, most applications can use a stale value without any changes.

3.2.2 Memcache Pools

Using memcache as a general-purpose caching layer requires workloads to share infrastructure despite different access patterns, memory footprints, and quality-of-service requirements. Different applications' workloads can produce negative interference resulting in decreased hit rates.

To accommodate these differences, we partition a cluster's memcached servers into separate pools. We designate one pool (named *wildcard*) as the default and provision separate pools for keys whose residence in wildcard is problematic. For example, we may provision a small pool for keys that are accessed frequently but for which a cache miss is inexpensive. We may also provision a large pool for infrequently accessed keys for which cache misses are prohibitively expensive.

Figure 5 shows the working set of two different sets of items, one that is low-churn and another that is high-churn. The working set is approximated by sampling all operations on one out of every one million items. For each of these items, we collect the minimum, average, and maximum item size. These sizes are summed and multiplied by one million to approximate the working set. The difference between the daily and weekly working sets indicates the amount of churn. Items with different churn characteristics interact in an unfortunate way: low-churn keys that are still valuable are evicted before high-churn keys that are no longer being accessed. Placing these keys in different pools prevents this kind of negative interference, and allows us to size high-churn pools appropriate to their cache miss cost. Section 7 provides further analysis.

3.2.3 Replication Within Pools

Within some pools, we use replication to improve the latency and efficiency of memcached servers. We choose to replicate a category of keys within a pool when (1) the application routinely fetches many keys simultaneously, (2) the entire data set fits in one or two memcached servers and (3) the request rate is much higher than what a single server can manage.

We favor replication in this instance over further dividing the key space. Consider a memcached server holding 100 items and capable of responding to 500k requests per second. Each request asks for 100 keys. The difference in memcached overhead for retrieving 100 keys per request instead of 1 key is small. To scale the system to process 1M requests/sec, suppose that we add a second server and split the key space equally between the two. Clients now need to split each request for 100 keys into two parallel requests for $\sim$50 keys. Consequently, both servers still have to process 1M requests per second. However, if we replicate all 100 keys to multiple servers, a client's request for 100 keys can be sent to any replica. This reduces the load per server to 500k requests per second. Each client chooses replicas based on its own IP address. This approach requires delivering invalidations to all replicas to maintain consistency.

3.3 Handling Failures

The inability to fetch data from memcache results in excessive load to backend services that could cause further cascading failures. There are two scales at which we must address failures: (1) a small number of hosts are inaccessible due to a network or server failure or (2) a widespread outage that affects a significant percentage of the servers within the cluster. If an entire cluster has to be taken offline, we divert user web requests to other clusters which effectively removes all the load from memcache within that cluster.

For small outages we rely on an automated remediation system [3]. These actions are not instant and can take up to a few minutes. This duration is long enough to cause the aforementioned cascading failures and thus we introduce a mechanism to further insulate backend services from failures. We dedicate a small set of machines, named *Gutter*, to take over the responsibilities of a few failed servers. Gutter accounts for approximately 1% of the memcached servers in a cluster.

When a memcached client receives no response to its get request, the client assumes the server has failed and issues the request again to a special *Gutter* pool. If this second request misses, the client will insert the appropriate key-value pair into the Gutter machine after querying the database. Entries in Gutter expire quickly to obviate

Gutter invalidations. Gutter limits the load on backend services at the cost of slightly stale data.

Note that this design differs from an approach in which a client rehashes keys among the remaining `memcached` servers. Such an approach risks cascading failures due to non-uniform key access frequency. For example, a single key can account for 20% of a server's requests. The server that becomes responsible for this hot key might also become overloaded. By shunting load to idle servers we limit that risk.

Ordinarily, each failed request results in a hit on the backing store, potentially overloading it. By using Gutter to store these results, a substantial fraction of these failures are converted into hits in the gutter pool thereby reducing load on the backing store. In practice, this system reduces the rate of client-visible failures by 99% and converts 10%–25% of failures into hits each day. If a `memcached` server fails entirely, hit rates in the gutter pool generally exceed 35% in under 4 minutes and often approach 50%. Thus when a few `memcached` servers are unavailable due to failure or minor network incidents, Gutter protects the backing store from a surge of traffic.

4 In a Region: Replication

It is tempting to buy more web and `memcached` servers to scale a cluster as demand increases. However, naïvely scaling the system does not eliminate all problems. Highly requested items will only become more popular as more web servers are added to cope with increased user traffic. Incast congestion also worsens as the number of `memcached` servers increases. We therefore split our web and `memcached` servers into multiple *frontend clusters*. These clusters, along with a storage cluster that contain the databases, define a *region*. This region architecture also allows for smaller failure domains and a tractable network configuration. We trade replication of data for more independent failure domains, tractable network configuration, and a reduction of incast congestion.

This section analyzes the impact of multiple frontend clusters that share the same storage cluster. Specifically we address the consequences of allowing data replication across these clusters and the potential memory efficiencies of disallowing this replication.

4.1 Regional Invalidations

While the storage cluster in a region holds the authoritative copy of data, user demand may replicate that data into frontend clusters. The storage cluster is responsible for invalidating cached data to keep frontend clusters consistent with the authoritative versions. As an optimization, a web server that modifies data also sends invalidations to its own cluster to provide read-after-write semantics for a single user request and reduce the amount of time stale data is present in its local cache.

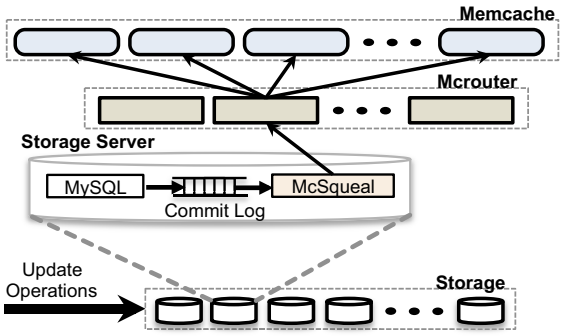


Figure 6: Invalidation pipeline showing keys that need to be deleted via the daemon (`mcsqueal`).

SQL statements that modify authoritative state are amended to include `memcache` keys that need to be invalidated once the transaction commits [7]. We deploy invalidation daemons (named `mcsqueal`) on every database. Each daemon inspects the SQL statements that its database commits, extracts any deletes, and broadcasts these deletes to the `memcache` deployment in every frontend cluster in that region. Figure 6 illustrates this approach. We recognize that most invalidations do not delete data; indeed, only 4% of all deletes issued result in the actual invalidation of cached data.

Reducing packet rates: While `mcsqueal` could contact `memcached` servers directly, the resulting rate of packets sent from a backend cluster to frontend clusters would be unacceptably high. This packet rate problem is a consequence of having many databases and many `memcached` servers communicating across a cluster boundary. Invalidation daemons batch deletes into fewer packets and send them to a set of dedicated servers running `mcrouter` instances in each frontend cluster. These `mcrouter`s then unpack individual deletes from each batch and route those invalidations to the right `memcached` server co-located within the frontend cluster. The batching results in an 18× improvement in the median number of deletes per packet.

Invalidation via web servers: It is simpler for web servers to broadcast invalidations to all frontend clusters. This approach unfortunately suffers from two problems. First, it incurs more packet overhead as web servers are less effective at batching invalidations than `mcsqueal` pipeline. Second, it provides little recourse when a systemic invalidation problem arises such as misrouting of deletes due to a configuration error. In the past, this would often require a rolling restart of the entire `memcache` infrastructure, a slow and disruptive pro-

	A (Cluster)	B (Region)
Median number of users	30	1
Gets per second	3.26 M	458 K
Median value size	10.7 kB	4.34 kB

Table 1: Deciding factors for cluster or regional replication of two item families

cess we want to avoid. In contrast, embedding invalidations in SQL statements, which databases commit and store in reliable logs, allows `mcsqueal` to simply replay invalidations that may have been lost or misrouted.

4.2 Regional Pools

Each cluster independently caches data depending on the mix of the user requests that are sent to it. If users' requests are randomly routed to all available frontend clusters then the cached data will be roughly the same across all the frontend clusters. This allows us to take a cluster offline for maintenance without suffering from reduced hit rates. Over-replicating the data can be memory inefficient, especially for large, rarely accessed items. We can reduce the number of replicas by having multiple frontend clusters share the same set of `memcached` servers. We call this a *regional pool*.

Crossing cluster boundaries incurs more latency. In addition, our networks have 40% less average available bandwidth over cluster boundaries than within a single cluster. Replication trades more `memcached` servers for less inter-cluster bandwidth, lower latency, and better fault tolerance. For some data, it is more cost efficient to forgo the advantages of replicating data and have a single copy per region. One of the main challenges of scaling `memcache` within a region is deciding whether a key needs to be replicated across all frontend clusters or have a single replica per region. Gutter is also used when servers in regional pools fail.

Table 1 summarizes two kinds of items in our application that have large values. We have moved one kind (B) to a regional pool while leaving the other (A) untouched. Notice that clients access items falling into category B an order of magnitude less than those in category A. Category B's low access rate makes it a prime candidate for a regional pool since it does not adversely impact inter-cluster bandwidth. Category B would also occupy 25% of each cluster's wildcard pool so regionalization provides significant storage efficiencies. Items in category A, however, are twice as large and accessed much more frequently, disqualifying themselves from regional consideration. The decision to migrate data into regional pools is currently based on a set of manual heuristics based on access rates, data set size, and number of unique users accessing particular items.

4.3 Cold Cluster Warmup

When we bring a new cluster online, an existing one fails, or perform scheduled maintenance the caches will have very poor hit rates diminishing the ability to insulate backend services. A system called *Cold Cluster Warmup* mitigates this by allowing clients in the "cold cluster" (*i.e.* the frontend cluster that has an empty cache) to retrieve data from the "warm cluster" (*i.e.* a cluster that has caches with normal hit rates) rather than the persistent storage. This takes advantage of the aforementioned data replication that happens across frontend clusters. With this system cold clusters can be brought back to full capacity in a few hours instead of a few days.

Care must be taken to avoid inconsistencies due to race conditions. For example, if a client in the cold cluster does a database update, and a subsequent request from another client retrieves the stale value from the warm cluster before the warm cluster has received the invalidation, that item will be indefinitely inconsistent in the cold cluster. `Memcached` deletes support nonzero hold-off times that reject `add` operations for the specified hold-off time. By default, all deletes to the cold cluster are issued with a two second hold-off. When a miss is detected in the cold cluster, the client re-requests the key from the warm cluster and `adds` it into the cold cluster. The failure of the `add` indicates that newer data is available on the database and thus the client will refetch the value from the databases. While there is still a theoretical possibility that deletes get delayed more than two seconds, this is not true for the vast majority of the cases. The operational benefits of cold cluster warmup far outweigh the cost of rare cache consistency issues. We turn it off once the cold cluster's hit rate stabilizes and the benefits diminish.

5 Across Regions: Consistency

There are several advantages to a broader geographic placement of data centers. First, putting web servers closer to end users can significantly reduce latency. Second, geographic diversity can mitigate the effects of events such as natural disasters or massive power failures. And third, new locations can provide cheaper power and other economic incentives. We obtain these advantages by deploying to multiple regions. Each region consists of a storage cluster and several frontend clusters. We designate one region to hold the master databases and the other regions to contain read-only replicas; we rely on MySQL's replication mechanism to keep replica databases up-to-date with their masters. In this design, web servers experience low latency when accessing either the local `memcached` servers or the local database replicas. When scaling across mul-

tiple regions, maintaining consistency between data in memcache and the persistent storage becomes the primary technical challenge. These challenges stem from a single problem: replica databases may lag behind the master database.

Our system represents just one point in the wide spectrum of consistency and performance trade-offs. The consistency model, like the rest of the system, has evolved over the years to suit the scale of the site. It mixes what can be practically built without sacrificing our high performance requirements. The large volume of data that the system manages implies that any minor changes that increase network or storage requirements have non-trivial costs associated with them. Most ideas that provide stricter semantics rarely leave the design phase because they become prohibitively expensive. Unlike many systems that are tailored to an existing use case, memcache and Facebook were developed together. This allowed the applications and systems engineers to work together to find a model that is sufficiently easy for the application engineers to understand yet performant and simple enough for it to work reliably at scale. We provide best-effort eventual consistency but place an emphasis on performance and availability. Thus the system works very well for us in practice and we think we have found an acceptable trade-off.

Writes from a master region: Our earlier decision requiring the storage cluster to invalidate data via daemons has important consequences in a multi-region architecture. In particular, it avoids a race condition in which an invalidation arrives *before* the data has been replicated from the master region. Consider a web server in the master region that has finished modifying a database and seeks to invalidate now stale data. Sending invalidations within the master region is safe. However, having the web server invalidate data in a replica region may be premature as the changes may not have been propagated to the replica databases yet. Subsequent queries for the data from the replica region will race with the replication stream thereby increasing the probability of setting stale data into memcache. Historically, we implemented mcsqueal after scaling to multiple regions.

Writes from a non-master region: Now consider a user who updates his data from a non-master region when replication lag is excessively large. The user's next request could result in confusion if his recent change is missing. A cache refill from a replica's database should only be allowed after the replication stream has caught up. Without this, subsequent requests could result in the replica's stale data being fetched and cached.

We employ a *remote marker* mechanism to minimize the probability of reading stale data. The presence of the marker indicates that data in the local replica database are potentially stale and the query should be redirected to the master region. When a web server wishes to update data that affects a key k, that server (1) sets a remote marker r_k in the region, (2) performs the write to the master embedding k and r_k to be invalidated in the SQL statement, and (3) deletes k in the local cluster. On a subsequent request for k, a web server will be unable to find the cached data, check whether r_k exists, and direct its query to the master or local region depending on the presence of r_k. In this situation, we explicitly trade additional latency when there is a cache miss, for a decreased probability of reading stale data.

We implement remote markers by using a regional pool. Note that this mechanism may reveal stale information during concurrent modifications to the same key as one operation may delete a remote marker that should remain present for another in-flight operation. It is worth highlighting that our usage of memcache for remote markers departs in a subtle way from caching results. As a cache, deleting or evicting keys is always a safe action; it may induce more load on databases, but does not impair consistency. In contrast, the presence of a remote marker helps distinguish whether a non-master database holds stale data or not. In practice, we find both the eviction of remote markers and situations of concurrent modification to be rare.

Operational considerations: Inter-region communication is expensive since data has to traverse large geographical distances (*e.g.* across the continental United States). By sharing the same channel of communication for the delete stream as the database replication we gain network efficiency on lower bandwidth connections.

The aforementioned system for managing deletes in Section 4.1 is also deployed with the replica databases to broadcast the deletes to memcached servers in the replica regions. Databases and mcrouters buffer deletes when downstream components become unresponsive. A failure or delay in any of the components results in an increased probability of reading stale data. The buffered deletes are replayed once these downstream components are available again. The alternatives involve taking a cluster offline or over-invalidating data in frontend clusters when a problem is detected. These approaches result in more disruptions than benefits given our workload.

6 Single Server Improvements

The *all-to-all* communication pattern implies that a single server can become a bottleneck for a cluster. This section describes performance optimizations and memory efficiency gains in memcached which allow better

scaling within clusters. Improving single server cache performance is an active research area [9, 10, 28, 25].

6.1 Performance Optimizations

We began with a single-threaded `memcached` which used a fixed-size hash table. The first major optimizations were to: (1) allow automatic expansion of the hash table to avoid look-up times drifting to $O(n)$, (2) make the server multi-threaded using a global lock to protect multiple data structures, and (3) giving each thread its own UDP port to reduce contention when sending replies and later spreading interrupt processing overhead. The first two optimizations were contributed back to the open source community. The remainder of this section explores further optimizations that are not yet available in the open source version.

Our experimental hosts have an Intel Xeon CPU (X5650) running at 2.67GHz (12 cores and 12 hyperthreads), an Intel 82574L gigabit ethernet controller and 12GB of memory. Production servers have additional memory. Further details have been previously published [4]. The performance test setup consists of fifteen clients generating `memcache` traffic to a single `memcached` server with 24 threads. The clients and server are co-located on the same rack and connected through gigabit ethernet. These tests measure the latency of `memcached` responses over two minutes of sustained load.

Get Performance: We first investigate the effect of replacing our original multi-threaded single-lock implementation with fine-grained locking. We measured hits by pre-populating the cache with 32-byte values before issuing `memcached` requests of 10 keys each. Figure 7 shows the maximum request rates that can be sustained with sub-millisecond average response times for different versions of `memcached`. The first set of bars is our `memcached` before fine-grained locking, the second set is our current `memcached`, and the final set is the open source version 1.4.10 which independently implements a coarser version of our locking strategy.

Employing fine-grained locking triples the peak get rate for hits from 600k to 1.8M items per second. Performance for misses also increased from 2.7M to 4.5M items per second. Hits are more expensive because the return value has to be constructed and transmitted, while misses require a single static response (`END`) for the entire multiget indicating that all keys missed.

We also investigated the performance effects of using UDP instead of TCP. Figure 8 shows the peak request rate we can sustain with average latencies of less than one millisecond for single gets and multigets of 10 keys. We found that our UDP implementation outperforms our TCP implementation by 13% for single gets and 8% for 10-key multigets.

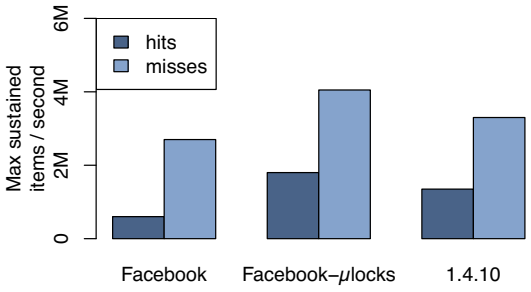


Figure 7: Multiget hit and miss performance comparison by `memcached` version

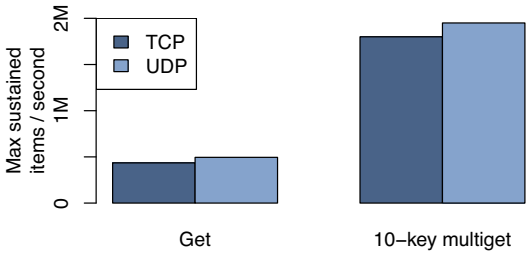


Figure 8: Get hit performance comparison for single gets and 10-key multigets over TCP and UDP

Because multigets pack more data into each request than single gets, they use fewer packets to do the same work. Figure 8 shows an approximately four-fold improvement for 10-key multigets over single gets.

6.2 Adaptive Slab Allocator

`Memcached` employs a slab allocator to manage memory. The allocator organizes memory into *slab classes*, each of which contains pre-allocated, uniformly sized chunks of memory. `Memcached` stores items in the smallest possible slab class that can fit the item's metadata, key, and value. Slab classes start at 64 bytes and exponentially increase in size by a factor of 1.07 up to 1 MB, aligned on 4-byte boundaries[3]. Each slab class maintains a free-list of available chunks and requests more memory in 1MB slabs when its free-list is empty. Once a `memcached` server can no longer allocate free memory, storage for new items is done by evicting the least recently used (LRU) item within that slab class. When workloads change, the original memory allocated to each slab class may no longer be enough resulting in poor hit rates.

[3]This scaling factor ensures that we have both 64 and 128 byte items which are more amenable to hardware cache lines.

We implemented an adaptive allocator that periodically re-balances slab assignments to match the current workload. It identifies slab classes as needing more memory if they are currently evicting items and if the next item to be evicted was used at least 20% more recently than the average of the least recently used items in other slab classes. If such a class is found, then the slab holding the least recently used item is freed and transferred to the needy class. Note that the open-source community has independently implemented a similar allocator that balances the eviction rates across slab classes while our algorithm focuses on balancing the age of the oldest items among classes. Balancing age provides a better approximation to a single global Least Recently Used (LRU) eviction policy for the entire server rather than adjusting eviction rates which can be heavily influenced by access patterns.

6.3 The Transient Item Cache

While `memcached` supports expiration times, entries may live in memory well after they have expired. `Memcached` lazily evicts such entries by checking expiration times when serving a get request for that item or when they reach the end of the LRU. Although efficient for the common case, this scheme allows short-lived keys that see a single burst of activity to waste memory until they reach the end of the LRU.

We therefore introduce a hybrid scheme that relies on lazy eviction for most keys and proactively evicts short-lived keys when they expire. We place short-lived items into a circular buffer of linked lists (indexed by seconds until expiration) – called the *Transient Item Cache* – based on the expiration time of the item. Every second, all of the items in the bucket at the head of the buffer are evicted and the head advances by one. When we added a short expiration time to a heavily used set of keys whose items have short useful lifespans; the proportion of `memcache` pool used by this key family was reduced from 6% to 0.3% without affecting the hit rate.

6.4 Software Upgrades

Frequent software changes may be needed for upgrades, bug fixes, temporary diagnostics, or performance testing. A `memcached` server can reach 90% of its peak hit rate within a few hours. Consequently, it can take us over 12 hours to upgrade a set of `memcached` servers as the resulting database load needs to be managed carefully. We modified `memcached` to store its cached values and main data structures in System V shared memory regions so that the data can remain live across a software upgrade and thereby minimize disruption.

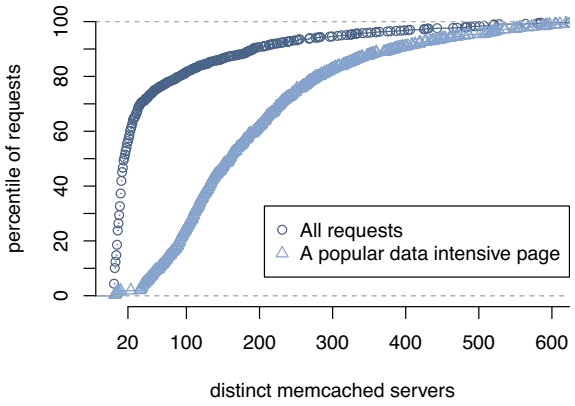


Figure 9: Cumulative distribution of the number of distinct `memcached` servers accessed

7 Memcache Workload

We now characterize the `memcache` workload using data from servers that are running in production.

7.1 Measurements at the Web Server

We record all `memcache` operations for a small percentage of user requests and discuss the fan-out, response size, and latency characteristics of our workload.

Fanout: Figure 9 shows the distribution of distinct `memcached` servers a web server may need to contact when responding to a page request. As shown, 56% of all page requests contact fewer than 20 `memcached` servers. By volume, user requests tend to ask for small amounts of cached data. There is, however, a long tail to this distribution. The figure also depicts the distribution for one of our more popular pages that better exhibits the all-to-all communication pattern. Most requests of this type will access over 100 distinct servers; accessing several hundred `memcached` servers is not rare.

Response size: Figure 10 shows the response sizes from `memcache` requests. The difference between the median (135 bytes) and the mean (954 bytes) implies that there is a very large variation in the sizes of the cached items. In addition there appear to be three distinct peaks at approximately 200 bytes and 600 bytes. Larger items tend to store lists of data while smaller items tend to store single pieces of content.

Latency: We measure the round-trip latency to request data from `memcache`, which includes the cost of routing the request and receiving the reply, network transfer time, and the cost of deserialization and decompression. Over 7 days the median request latency is 333 microseconds while the 75^{th} and 95^{th} percentiles (p75 and p95) are 475μs and 1.135ms respectively. Our median end-to-end latency from an idle web server is 178μs while the p75 and p95 are 219μs and 374μs, respectively. The

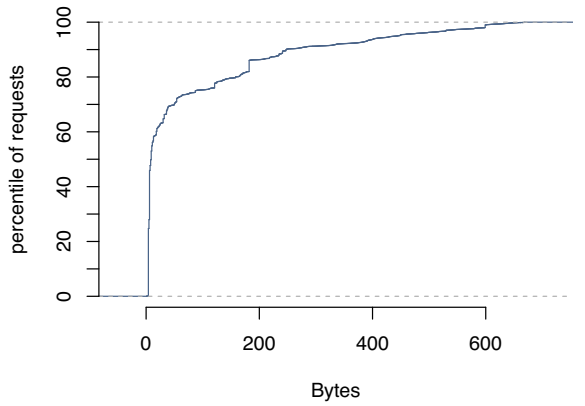


Figure 10: Cumulative distribution of value sizes fetched

wide variance between the p95 latencies arises from handling large responses and waiting for the runnable thread to be scheduled as discussed in Section 3.1.

7.2 Pool Statistics

We now discuss key metrics of four `memcache` pools. The pools are wildcard (the default pool), app (a pool devoted for a specific application), a replicated pool for frequently accessed data, and a regional pool for rarely accessed information. In each pool, we collect average statistics every 4 minutes and report in Table 2 the highest average for one month collection period. This data approximates the peak load seen by those pools. The table shows the widely different get, set, and delete rates for different pools. Table 3 shows the distribution of response sizes for each pool. Again, the different characteristics motivate our desire to segregate these workloads from one another.

As discussed in Section 3.2.3, we replicate data within a pool and take advantage of batching to handle the high request rates. Observe that the replicated pool has the highest get rate (about 2.7× that of the next highest one) and the highest ratio of bytes to packets despite having the smallest item sizes. This data is consistent with our design in which we leverage replication and batching to achieve better performance. In the *app* pool, a higher churn of data results in a naturally higher miss rate. This pool tends to have content that is accessed for a few hours and then fades away in popularity in favor of newer content. Data in the regional pool tends to be large and infrequently accessed as shown by the request rates and the value size distribution.

7.3 Invalidation Latency

We recognize that the timeliness of invalidations is a critical factor in determining the probability of exposing stale data. To monitor this health, we sample one out of a million deletes and record the time the delete was issued. We subsequently query the contents of `memcache` across all frontend clusters at regular intervals for the sampled keys and log an error if an item remains cached despite a delete that should have invalidated it.

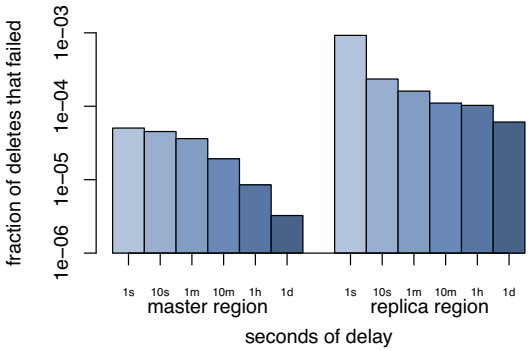


Figure 11: Latency of the Delete Pipeline

In Figure 11, we use this monitoring mechanism to report our invalidation latencies across a 30 day span. We break this data into two different components: (1) the delete originated from a web server in the master region and was destined to a `memcached` server in the master region and (2) the delete originated from a replica region and was destined to another replica region. As the data show, when the source and destination of the delete are co-located with the master our success rates are much higher and achieve four 9s of reliability within 1 second and five 9s after one hour. However when the deletes originate and head to locations outside of the master region our reliability drops to three 9s within a second and four 9s within 10 minutes. In our experience, we find that if an invalidation is missing after only a few seconds the most common reason is that the first attempt failed and subsequent retrials will resolve the problem.

8 Related Work

Several other large websites have recognized the utility of key-value stores. DeCandia *et al.* [12] present a highly available key-value store that is used by a variety of application services at Amazon.com. While their system is optimized for a write heavy workload, ours targets a workload dominated by reads. Similarly, LinkedIn uses Voldemort [5], a system inspired by Dynamo. Other major deployments of key-value caching solutions include Redis [6] at Github, Digg, and Blizzard, and `memcached` at Twitter [33] and Zynga. Lakshman *et al.* [1] developed Cassandra, a schema-based distributed key-value store. We preferred to deploy and scale `memcached` due to its simpler design.

Our work in scaling `memcache` builds on extensive work in distributed data structures. Gribble *et al.* [19]

pool	miss rate	$\frac{get}{s}$	$\frac{set}{s}$	$\frac{delete}{s}$	$\frac{packets}{s}$	outbound bandwidth (MB/s)
wildcard	1.76%	262k	8.26k	21.2k	236k	57.4
app	7.85%	96.5k	11.9k	6.28k	83.0k	31.0
replicated	0.053%	710k	1.75k	3.22k	44.5k	30.1
regional	6.35%	9.1k	0.79k	35.9k	47.2k	10.8

Table 2: Traffic per server on selected `memcache` pools averaged over 7 days

pool	mean	std dev	p5	p25	p50	p75	p95	p99
wildcard	1.11 K	8.28 K	77	102	169	363	3.65 K	18.3 K
app	881	7.70 K	103	247	269	337	1.68K	10.4 K
replicated	66	2	62	68	68	68	68	68
regional	31.8 K	75.4 K	231	824	5.31 K	24.0 K	158 K	381 K

Table 3: Distribution of item sizes for various pools in bytes

present an early version of a key-value storage system useful for Internet scale services. Ousterhout *et al.* [29] also present the case for a large scale in-memory key-value storage system. Unlike both of these solutions, `memcache` does not guarantee persistence. We rely on other systems to handle persistent data storage.

Ports *et al.* [31] provide a library to manage the cached results of queries to a transactional database. Our needs require a more flexible caching strategy. Our use of leases [18] and stale reads [23] leverages prior research on cache consistency and read operations in high-performance systems. Work by Ghandeharizadeh and Yap [15] also presents an algorithm that addresses the stale set problem based on time-stamps rather than explicit version numbers.

While software routers are easier to customize and program, they are often less performant than their hardware counterparts. Dobrescu *et al.* [13] address these issues by taking advantage of multiple cores, multiple memory controllers, multi-queue networking interfaces, and batch processing on general purpose servers. Applying these techniques to `mcrouter`'s implementation remains future work. Twitter has also independently developed a `memcache` proxy similar to `mcrouter` [32].

In Coda [35], Satyanarayanan *et al.* demonstrate how datasets that diverge due to disconnected operation can be brought back into sync. Glendenning *et al.* [17] leverage Paxos [24] and quorums [16] to build Scatter, a distributed hash table with linearizable semantics [21] resilient to churn. Lloyd *et al.* [27] examine causal consistency in COPS, a wide-area storage system.

TAO [37] is another Facebook system that relies heavily on caching to serve large numbers of low-latency queries. TAO differs from `memcache` in two fundamental ways. (1) TAO implements a graph data model in which nodes are identified by fixed-length persistent identifiers (64-bit integers). (2) TAO encodes a specific mapping of its graph model to persistent storage and takes responsibility for persistence. Many components, such as our client libraries and `mcrouter`, are used by both systems.

9 Conclusion

In this paper, we show how to scale a `memcached`-based architecture to meet the growing demand of Facebook. Many of the trade-offs discussed are not fundamental, but are rooted in the realities of balancing engineering resources while evolving a live system under continuous product development. While building, maintaining, and evolving our system we have learned the following lessons. (1) Separating cache and persistent storage systems allows us to independently scale them. (2) Features that improve monitoring, debugging and operational efficiency are as important as performance. (3) Managing stateful components is operationally more complex than stateless ones. As a result keeping logic in a stateless client helps iterate on features and minimize disruption. (4) The system must support gradual rollout and rollback of new features even if it leads to temporary heterogeneity of feature sets. (5) Simplicity is vital.

Acknowledgements

We would like to thank Philippe Ajoux, Nathan Bronson, Mark Drayton, David Fetterman, Alex Gartrell, Andrii Grynenko, Robert Johnson, Sanjeev Kumar, Anton Likhtarov, Mark Marchukov, Scott Marlette, Ben Maurer, David Meisner, Konrad Michels, Andrew Pope, Jeff Rothschild, Jason Sobel, and Yee Jiun Song for their contributions. We would also like to thank the anonymous reviewers, our shepherd Michael Piatek, Tor M. Aamodt, Remzi H. Arpaci-Dusseau, and Tayler Hetherington for their valuable feedback on earlier drafts of the paper. Finally we would like to thank our fellow engineers at Facebook for their suggestions, bug-reports, and support which makes `memcache` what it is today.

References

[1] Apache Cassandra. http://cassandra.apache.org/.

[2] Couchbase. http://www.couchbase.com/.

[3] Making Facebook Self-Healing. https://www.facebook.com/note.php?note_id=10150275248698920.

[4] Open Compute Project. http://www.opencompute.org.

[5] Project Voldemort. http://project-voldemort.com/.

[6] Redis. http://redis.io/.

[7] Scaling Out. https://www.facebook.com/note.php?note_id=23844338919.

[8] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. *ACM SIGMETRICS Performance Evaluation Review 40*, 1 (June 2012), 53–64.

[9] BEREZECKI, M., FRACHTENBERG, E., PALECZNY, M., AND STEELE, K. Power and performance evaluation of memcached on the tilepro64 architecture. *Sustainable Computing: Informatics and Systems 2*, 2 (June 2012), 81 – 90.

[10] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design & Implementation* (2010), pp. 1–8.

[11] CERF, V. G., AND KAHN, R. E. A protocol for packet network intercommunication. *ACM SIGCOMM Compututer Communication Review 35*, 2 (Apr. 2005), 71–82.

[12] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review 41*, 6 (Dec. 2007), 205–220.

[13] FALL, K., IANNACCONE, G., MANESH, M., RATNASAMY, S., ARGYRAKI, K., DOBRESCU, M., AND EGI, N. Routebricks: enabling general purpose network infrastructure. *ACM SIGOPS Operating Systems Review 45*, 1 (Feb. 2011), 112–125.

[14] FITZPATRICK, B. Distributed caching with memcached. *Linux Journal 2004*, 124 (Aug. 2004), 5.

[15] GHANDEHARIZADEH, S., AND YAP, J. Gumball: a race condition prevention technique for cache augmented sql database management systems. In *Proceedings of the 2nd ACM SIGMOD Workshop on Databases and Social Networks* (2012), pp. 1–6.

[16] GIFFORD, D. K. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles* (1979), pp. 150–162.

[17] GLENDENNING, L., BESCHASTNIKH, I., KRISHNAMURTHY, A., AND ANDERSON, T. Scalable consistency in Scatter. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), pp. 15–28.

[18] GRAY, C., AND CHERITON, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review 23*, 5 (Nov. 1989), 202–210.

[19] GRIBBLE, S. D., BREWER, E. A., HELLERSTEIN, J. M., AND CULLER, D. Scalable, distributed data structures for internet service construction. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design & Implementation* (2000), pp. 319–332.

[20] HEINRICH, J. *MIPS R4000 Microprocessor User's Manual*. MIPS technologies, 1994.

[21] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12*, 3 (July 1990), 463–492.

[22] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. Consistent Hashing and Random trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the 29th annual ACM Symposium on Theory of Computing* (1997), pp. 654–663.

[23] KEETON, K., MORREY, III, C. B., SOULES, C. A., AND VEITCH, A. Lazybase: freshness vs. performance in information management. *ACM SIGOPS Operating Systems Review 44*, 1 (Dec. 2010), 15–19.

[24] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[25] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Silt: a memory-efficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), pp. 1–13.

[26] LITTLE, J., AND GRAVES, S. Little's law. *Building Intuition* (2008), 81–100.

[27] LLOYD, W., FREEDMAN, M., KAMINSKY, M., AND ANDERSEN, D. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), pp. 401–416.

[28] METREVELI, Z., ZELDOVICH, N., AND KAASHOEK, M. Cphash: A cache-partitioned hash table. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (2012), pp. 319–320.

[29] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., ONGARO, D., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for ramcloud. *Communications of the ACM 54*, 7 (July 2011), 121–130.

[30] PHANISHAYEE, A., KREVAT, E., VASUDEVAN, V., ANDERSEN, D. G., GANGER, G. R., GIBSON, G. A., AND SESHAN, S. Measurement and analysis of tcp throughput collapse in cluster-based storage systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), pp. 12:1–12:14.

[31] PORTS, D. R. K., CLEMENTS, A. T., ZHANG, I., MADDEN, S., AND LISKOV, B. Transactional consistency and automatic management in an application data cache. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design & Implementation* (2010), pp. 1–15.

[32] RAJASHEKHAR, M. Twemproxy: A fast, light-weight proxy for memcached. https://dev.twitter.com/blog/twemproxy.

[33] RAJASHEKHAR, M., AND YUE, Y. Caching with twemcache. http://engineering.twitter.com/2012/07/caching-with-twemcache.html.

[34] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. *ACM SIGCOMM Computer Communication Review 31*, 4 (Oct. 2001), 161–172.

[35] SATYANARAYANAN, M., KISTLER, J., KUMAR, P., OKASAKI, M., SIEGEL, E., AND STEERE, D. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers 39*, 4 (Apr. 1990), 447–459.

[36] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review 31*, 4 (Oct. 2001), 149–160.

[37] VENKATARAMANI, V., AMSDEN, Z., BRONSON, N., CABRERA III, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., HOON, J., KULKARNI, S., LAWRENCE, N., MARCHUKOV, M., PETROV, D., AND PUZAR, L. Tao: how facebook serves the social graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2012), pp. 791–792.

F10: A Fault-Tolerant Engineered Network

Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, Thomas Anderson
University of Washington

Abstract

The data center network is increasingly a cost, reliability and performance bottleneck for cloud computing. Although multi-tree topologies can provide scalable bandwidth and traditional routing algorithms can provide eventual fault tolerance, we argue that recovery speed can be dramatically improved through the co-design of the network topology, routing algorithm and failure detector. We create an engineered network and routing protocol that directly address the failure characteristics observed in data centers. At the core of our proposal is a novel network topology that has many of the same desirable properties as FatTrees, but with much better fault recovery properties. We then create a series of failover protocols that benefit from this topology and are designed to cascade and complement each other. The resulting system, F10, can almost instantaneously reestablish connectivity and load balance, even in the presence of multiple failures. Our results show that following network link and switch failures, F10 has less than 1/7th the packet loss of current schemes. A trace-driven evaluation of MapReduce performance shows that F10's lower packet loss yields a median application-level 30% speedup.

1 Introduction

Data center networks are an increasingly important component to the cost, reliability and performance of cloud services. This has led to recent efforts by the network research community to explore new topologies [11, 12, 13], new routing protocols [11] and new network management layers [3, 4, 20], with a goal of improving network cost-effectiveness, fault tolerance and scalability.

A state of the art approach is taken by Al-Fares et al. [3] and its followup project PortLand [20]. In these systems, the data center network is constructed in a multi-rooted tree structure called a FatTree (inspired by fat-trees [17]) of inexpensive, commodity switches. These proposals provide scalability, both in terms of port count and the overall bisection bandwidth of the network. They also deliver better performance at low costs, primarily due to their use of commodity switches.

The use of a large number of commodity switches, however, opens up questions regarding what happens when links and switches fail. A FatTree has redundant paths between any pair of hosts. If end host operating system changes are possible between these end hosts, the network can be set up to provide multiple paths. The end host manages packet loss and congestion across the paths using MPTCP [22]. In many cases, the data center operator is not in control of the OS, requiring a network-level solution to fault tolerance. A consequence of our work is to show that entirely network-level failure recovery can be practical and nearly instantaneous in a data center setting.

Addressing this need for network-layer recovery, FatTree architectures have proposed using a centralized manager that collects topology and failure information from the switches. It then periodically generates and disseminates back to the switches and end-hosts alternate sets of routes to avoid failures. Centralized route management is both simple and flexible—a reasonable design choice provided that failures do not occur very often.

Recent measurements of network-layer failures in data centers, however, have shown that failures are frequent and disruptive [10]. Network-layer failures can reduce the volume of traffic delivered by more than 40%, even when the underlying network is designed for failure resilience. As data centers grow, the probability of network failures and the consequent disruptions on the system as a whole will likely increase, further exacerbating the problem.

Our goal is to co-design a topology and set of protocols that admit near-instantaneous, fine-grained, localized, network-level recovery and rebalancing for common-case network failures. Because the network is already a significant part of the cost of the data center, we limit ourselves to not introducing any additional hardware relative to PortLand. Other work has shown that local repair is possible at the cost of significant added hardware relative to a standard FatTree [9, 12, 13], so our work can be seen as either improving the speed of repair in FatTree and other multi-tree networks or in reducing the hardware cost of fast repair in more general networks. A limitation of our work is that we assume that we can change both the network topology and the protocols used between network switches.

Our system is called F10 (the Fault-Tolerant Engineered Network), a network topology and a set of protocols that can recover rapidly from almost all data center network failures. We design a novel topology to make it easier to do localized repair and rebalancing after failures. This topology is applicable to the FatTree and other multi-tree networks. We then redesign the routing protocols to take advantage of the modified topology. To satisfy the need for extremely fast failover, we use a local recovery mechanism that reacts almost instantaneously at the cost of additional latency and increased congestion. Some failures are not short-term, so local rerouting eventually triggers a slightly slower pushback mechanism that redirects traffic flows before they reach the faulty components.

To address longer-term failures, a centralized scheduler rearranges traffic on a much slower time scale in order to create as close to a optimally rerouted configuration as possible. We also introduce a failure detector that benefits from (and contributes to) the speed of our failover protocols while providing fine-grained information not available to traditional failure detection methods.

We have implemented a Click-based prototype of F10 and its failure detector and have performed a simulation-based evaluation, based on measurements of real-world data center traffic from [5] and measurements of data center network failures from [10]. Our results show that our system dramatically improves packet loss relative to PortLand with no added hardware cost. Our localized reroutes do incur some path inflation and network state, but these effects are small because of our novel topology.

2 Motivation

Our goal is to design a data center network architecture that can gracefully and quickly recover after failures, without any additional hardware. To motivate our approach, we outline previous measurements of data center network failures and then discuss the implications of these results on the design of fault-tolerant data center networks.

2.1 Failures in Data Centers

A recent study by Gill et al. provides insight into the characteristics of data center network failures [10]. The authors found that a large majority of devices are failure-free over the course of a year; commodity switches are mostly reliable. Their data also shows, however, that there are frequent short-term failures, that link failures are common and that the network responsiveness to failures is limited. We emphasize a few results from their study:

- **Many failures are short-term.** Devices and links exhibit a large number of short-term failures. In fact, the authors observed that the most failure-prone devices have a median time-to-failure of 8.6 minutes.

- **Multiple failures are common.** Devices often fail in groups. 41% of link failure events affect multiple devices—often, just a few (2–4) links, but in 10% of cases, they do affect more than 4 devices. There are also often multiple independent ongoing failures.

- **Some failures have long downtimes.** Though most failures are short-term, failure durations exhibit a long tail. Gill et al. attribute this to issues such as firmware bugs and device unreliability. Hardware that fails often stays down and contributes heavily to network-level unavailability.

- **Network faults impact network efficiency.** The data centers studied by Gill et al. have 1:1 redundancy dedicated to failure recovery, yet the network delivered only about 90% of the traffic in the median failure case. Performance is worse in the tail, with only 60% of traffic delivered during 20% of failures. This suggests better methods are needed for exploiting existing redundancy.

The authors assume a model where hardware is either up or down and transitions between those two states, but certain parts of their data—along with anecdotal evidence of *gray failures* from industry—conforms to a stochastic model of failures in which hardware loses a certain percentage of packets. There is thus an additional concern:

- **Existing failure detection mechanisms are too coarse-grained.** Links are marked as down after losing a certain number of heartbeats and marked as up after a brief handshake. Within a short time frame, it is difficult to distinguish between a complete failure, where no packets are getting through, and a situation where the link is congested, and had gotten unlucky with the heartbeats. Conversely, a flaky link that just happened to allow a handshake would appear to be reliable.

2.2 Next-Generation Data Center Networks

Today's data center networks are multi-level, multi-rooted trees of switches. The leaves of the tree are Top-of-Rack (ToR) switches that connect down to many machines in a rack, and up to the network core which aggregates and transfers traffic between racks. A modern data center might have racks that contain 40 servers connected with 1 Gbps access links, and one or two 10 Gbps uplinks that connect the ToR switch to the core, which contains a small number of significantly more expensive switches with an even faster interconnect. The primary challenges with these networks are that they do not *scale*—port counts and internal backplane bandwidth of core switches are limited and expensive—and that they are *dramatically oversubscribed*, with reported factors of 1:240 [11].

Recent proposals for the next generation of data center networks [3, 11, 12] overcome these limitations. We focus on a class of these networks based on the FatTree [3] proposal and its subsequent extensions. Inspired by the concept of a fat-tree [17], these FatTrees use a multi-rooted, multi-stage tree structure identical to a folded Clos network [15].[1] Just like in a fat-tree, child subtrees are stitched together at each level of the FatTree with thicker and thicker edges until there remains a single root tree, but unlike fat-trees, FatTrees can be built with uniformly-sized switches and links.

The benefit of these networks is that they can be made

[1] Since there are a few key distinctions between their instantiations, we clarify them here. We use fat-tree to denote the classical concept where links increase in capacity as you travel up toward the root. We use FatTree to denote the proposal of Al-Fares et al. [3], which uses multiple rooted trees to approximate a fat-tree. A similar caveat applies to the research literature's use of the terminology for Clos networks, which route messages along equal-length paths between distinct input and output terminals; folded Clos networks, which make no distinctions between terminals; and FatTrees, which allow short-circuiting of paths between nodes in a folded Clos network subtree.

of cheaper, commodity switches and provide much more path diversity within the network. PortLand takes advantage of this path diversity by using ECMP, which randomly places flows across physical paths. While ECMP lets us take advantage of the increased bandwidth provided by multiple paths, placing a flow on a single physical path means that failures will disrupt entire flows. An alternative is to upgrade the OS and let the end host use a protocol like MPTCP; however, it is not always the case that network operators have the ability to change end host OSes. In this paper, *we explore whether we can make network failures lightweight from the perspective of the end host* so that data center operators can run any end host system and not what is needed for the network.

To ease exposition, we will focus on a *non-oversubscribed* FatTree, in which half of the ports are used as downlinks to connect nodes within the same subtree, and half used as uplinks to access other parts of the tree. However, our system handles both *oversubscribed* (which allocate more ports to downlinks and can scale to more nodes or use few layers) and *overprovisioned* (which allocate more ports to uplinks for reliability and bisection bandwidth) variants, discussed further in Section 8. The root nodes, which do not have uplink edges, use all ports for downlinks. Figure 1a depicts a 3-level FatTree built from 4-port switches.

Our goal is near-instantaneous recovery from failures and load spikes with no added hardware. The original design of Clos networks was more concerned with non-blocking behavior than fault tolerance. Similarly, the papers introducing FatTrees and related proposals [3, 4, 20] discuss basic failover mechanisms, but are principally focused on achieving good bisection bandwidth with commodity switches [3], scalability, resilience to (but not rapid recovery from) faults [20], and centralized load-balancing [4]. These proposals are inherently limited in their ability to recover quickly and thoroughly from faults.

Limited local rerouting: While modern data centers have a variety of failover mechanisms, few are truly local. Data centers that use a link-state protocol such as OSPF require updates sent across the entire network before convergence. PortLand uses a centralized topology manager. VL2 [11] suggested detouring around a fault on the upward path, but it does not reroute around failures on the downward path because (as we explain below) there is only one path from any given root to a leaf switch.

Failure information must propagate to many and distant nodes: This deficiency goes beyond the lack of a suitable protocol. Consider Figure 1a.[2] No parent or grandparent of the failed node has any downlink path to the affected subtree. This property follows from the fat-tree-style construction that there is only ever one downlink path from the root of a subtree to any of its children. Among the nodes whose routes could reach a failed node, only those located *lower in the tree than the failure* have a route that avoids the failure. In other words, *no protocol that informs only nodes in the top portion of the tree will restore connectivity.* In the case of a failure on the downward portion of a path, any detour or pushback/broadcast protocol will be forced to travel from the parent of the failure all the way back to every node in the entire tree lower than the failure.

Irregular tree structure because of long-term faults: While data center operators aim to rapidly repair or replace failed equipment, as a practical matter, failures can persist for long periods of time. This can leave the system in a suboptimal state with poor load balancing. Multiple failures make this problem even worse. In our view, it is crucial that data center networks gracefully handle missing links and loss of symmetry. A negative example of this is the simple application of ECMP, which spreads load from a failed link to all remaining links at a local level, but does not evenly shift load to the remaining paths.

3 Design Overview

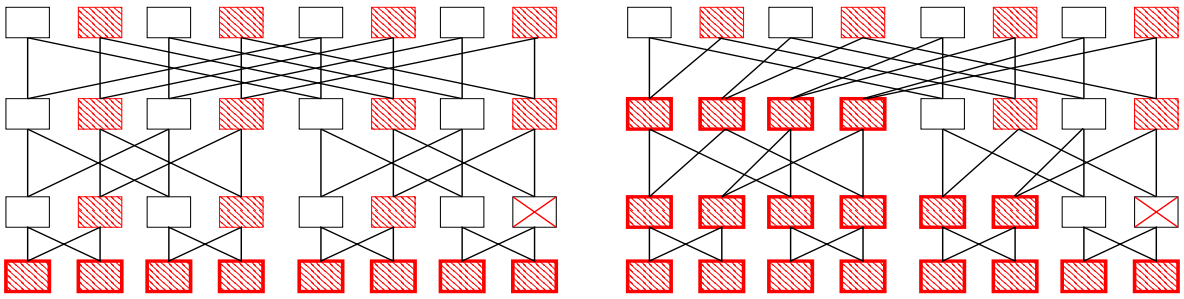
Taking the above concerns into account, we create an engineered network and routing protocol that can rapidly restore network connectivity and performance. Our system, F10, relies on the following ideas:

- **Planned asymmetry:** We propose a network topology that introduces a limited amount of asymmetry to achieve greater failure tolerance. The basic insight is that next-generation topologies provide many desirable properties, but there are variants that provide the same basic properties and are more resilient to failures.
- **Cascaded failover mechanisms:** Our system uses different mechanisms at different time scales to achieve short-term patching, medium-term fault avoidance, and longer-term load balancing.
- **Co-design of everything:** Each of the components of F10 (i.e., the topology, failover and load balancing protocols, and failure detector) are designed to enhance and support each other. As part of this approach, we unify the related problems of failure recovery and load balancing and use a similar set of mechanisms for both.

We elaborate on these design points below.

AB FatTree: We introduce a novel topology, the AB FatTree. By skewing the symmetry of a traditional FatTree, the AB FatTree allows for efficient local rerouting. The benefits come at almost no cost. The network requires no extra hardware, does not lose bisection bandwidth, and has similar properties to standard FatTrees (e.g., unique paths from a root to leaf, non-blocking behavior, etc.).

[2] For simplicity, we omit from several of our figures the doubled subtrees generated by folding the root uplinks into downlinks.



(a) (b)

Figure 1: Path alternatives in (a) a standard FatTree and (b) an AB FatTree. The X indicates a failure, and the hashed rectangles represent switches that are affected by it when trying to send to its children. Bold borders indicate affected switches that have a path around the failure. In the AB FatTree, more switches are affected, but more have alternatives, and they are closer to the failure.

Local rerouting: To satisfy the need for fast failover, we use a local recovery mechanism that is able to reroute the very next packet after failure detection. Because we fix the topology, we can design a purely local mechanism that is initiated and torn down at the affected switch and does not cause any convergence issues or broader disruptions.

Pushback notification: The reroute uses extra hops then the global optimum. Our system adds a slightly slower pushback mechanism that removes the additional latency, reducing the impact on congestion of local recovery.

Global re-optimization: On a much slower time scale, a centralized scheduler rearranges traffic to optimally balance load, despite failures.

Failure Detector: The lightweight and local nature of our failover protocols means that we can be more aggressive in marking links and switches as down, improving network performance. Our failure detector also provides and uses finer-grained information about the exact loss characteristics of the connection.

To accomplish the above, we assume a few things about the hardware. On the most basic level, we assume that we can modify the control plane of switches to execute our protocols locally and that switches can do local neighbor failure detection. We also assume the presence of a fault-tolerant controller and ability to readdress destinations with a location-based address, as in PortLand. For flow scheduling, we assume switches support consistent flow-based assignment for each source-destination pair. Our system can also benefit from the ability of switches to randomly place flows based on configured weights calculated by the central controller; however, this weighted placement is not essential for correct operation.

4 The AB FatTree

As we saw in Section 2.2, the standard FatTree design by Al-Fares et al. [3] has a structural weakness that makes it difficult to locally reroute around network failures. We introduce a novel topology, the AB FatTree, that skews the symmetry of a traditional FatTree to address this issue.

Notation	Definition or Value
k	# of ports per switch, e.g., 24
$L+1$	# of levels in the network, e.g., 3
p	$k/2$: # of up/downlinks per switch
N	$2p^{L+1}$: # of end hosts in the data center
b	$\lceil \log_2(p) \rceil$: # of bits per level in a node location
$prefix(a,i)$	$a \gg (ib)$: relevant prefix of location a at level i
$same\_prefix(a,a',i)$	$(prefix(a,i) \equiv prefix(a',i))$: whether a and a' share a prefix at level i

Table 1: A key to the notation used in this paper.

The key weakness in the standard FatTree is that all subtrees at level i are wired to the parents at level $i+1$ in an identical fashion. A parent attempting to detour around a failed child must use roundabout paths (with inflation of at least four hops) because all paths from its $p-1$ other children to the target subtree use the same failed node. The AB FatTree solves this problem by defining two types of subtrees (called *type A* and *type B*) that are wired to their parents in *two different ways*. With this simple change, a parent with a failed child in a type A subtree can detour to that subtree in two hops through the parents of a child in a type B subtree (and vice versa), because those parents *do not* rely on the failed node.

We now make the design concrete. Let k be the number of ports on each switch element, and L be the number of levels; as in the standard FatTree we use $p = k/2$ ports each for uplink and downlink at each switch, and can connect a total of $N = 2p^L$ end hosts in a rearrangeably non-blocking manner to the network. Table 1 contains a summary of the notation we use in this paper.

Figure 2 shows the labeled structure of an AB FatTree for $k = 4$ and $L = 3$, explained in the next few paragraphs.

Connectivity. For levels numbered 0 through L, each level $i < L$ contains $2p^L$ switches arranged in $2p^{L-i}$ groups of p^i switches.[3] Each group at level i represents a multi-rooted subtree of p^{i+1} end hosts with p^i root switches. The distinction between the standard version

[3]The top level ($i = L$) has one group of p^L switches, using all ports for downlinks.

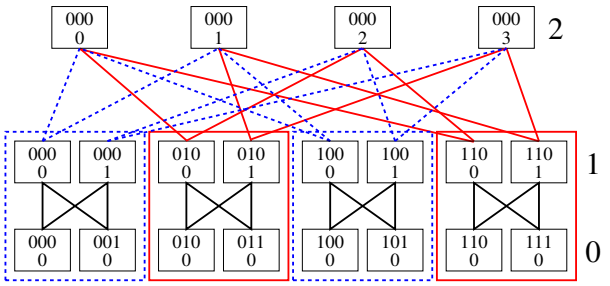


Figure 2: A labeled AB FatTree in which the subtrees with dotted blue lines are of type A and the subtrees with solid red lines are of type B. The numbers to the right of the tree are the *level*, the top number in each switch is the *location*, and the bottom number is the *index*.

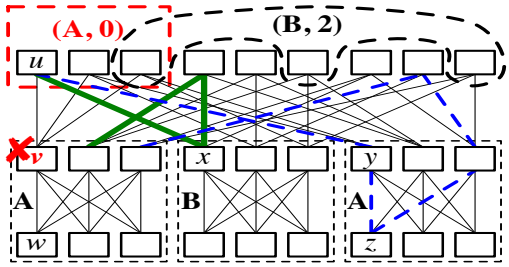


Figure 3: Illustration of the base cases of local rerouting with a failure at v. In the upward direction, w avoids v by routing to any other parent. Downward, u must find detours that avoid the failure group (A,0). The bold green path shows Scheme 1 rerouting through a type B child x, and the dotted blue path shows Scheme 2 rerouting through a child y of same type A.

and an AB FatTree is in the method of connecting these root switches to their parents.

Let j denote the index of a root node numbered 0 through $p^i - 1$ in level i. In a **type A** subtree, root j will be connected to the p *consecutive parents* numbered jp through $(j+1)p-1$. A standard FatTree contains only type A subtrees, whereas in an AB FatTree only half the subtrees are of type A. The remainder are of **type B**, wherein children connect to parents with a *stride of* p^i: root j is connected to parents j, $j+p^i$, $j+2p^i$, etc.

Addressing/Routing. A switch is uniquely identified by:

- *level* i – The level of the subtree of which it is a root.
- *index* j – The roots of a specific subtree are consecutively numbered as described above.
- *location* – The location of a node is an $Lb+1$-bit number constructed such that all nodes in the same level i subtree share a prefix of $(L-i)b+1$ bits that encodes the path from the root group to the subtree, where $b = \lceil \log_2 p \rceil$. The location has the format: ($b+1$ bits for level L).(b bits for level $L-1$)…(b bits for level $i+1$), concatenated with ib zero bits for levels i through 0.

In the absence of failures, routing occurs much like in PortLand [20]—each packet is routed upwards until it is able to travel back down, following longest-prefix matching. By construction, each subtree owns a single location address and the roots of a subtree can access one child in each of its subsubtrees. When a packet's destination lies within the subtree rooted in the current node, it will be routed downwards, otherwise it is forwarded upward.

Versus a standard FatTree. Revisiting Figure 1, we see that this rewiring allows nodes in subtrees of a different type to route around failures, in addition to nodes on a lower level that already had alternate paths. While the number of switches with affected paths increases, the total number of failed paths stays the same, and therefore the effects of the failure are distributed across more switches. As a consequence, more nodes have alternate paths, and there are alternatives closer to the failure.

5 Handling Failures

Our failover protocol consists of three stages that operate on increasing timescales. (1) When a switch detects a failure in one of its links, it immediately begins using *local rerouting* to reroute the very next packet. (2) Since local rerouting inflates paths as well as increases local congestion, the switch initiates a *pushback protocol* that causes upstream switches to redirect traffic to resume using shortest paths. (3) Finally, to deal with long-term failures that create a structural imbalance in the network, a *centralized rerouting* protocol determines an efficient global rearrangement of flows. In addition, the key to fast failure recovery is rapid and accurate failure detection, which is discussed at the end of this section.

5.1 Local Rerouting

Our first step after a failure is to quickly establish a new working route using only local information. We explain this using Figure 3, which shows a 3-level AB FatTree with $k = 6$. We label nodes u, v, and w, where v has failed.

Note that local rerouting for **upward links** in any multi-rooted tree is simple. A child (w) can route around a failed parent (v), by simply redirecting affected flows to any working parent. This restores connectivity without increasing the number of hops or requiring control traffic. In the unlikely event that all parents have failed, the child drops the packet; an alternative route will soon be configured by the pushback schemes discussed later unless the node is a leaf node. Most data center services are designed to tolerate rack-level failures. Alternatively, each leaf node can be wired into multiple ToR switches.

The rest of this section discusses rerouting of traffic for failed **downward links**. This case is significantly more complex, because when a child (v) fails, its parents (e.g., u) lose the only working path to that subtree (identified by $prefix(v)$) that follows standard routing policy. Instead, we propose two local detouring schemes. The first mechanism results in shorter detours, but $p/2$ failures located at

specific locations can cause it to fail. The second mechanism succeeds in more cases, but will have longer paths.

Scheme 1: three-hop rerouting. In most cases, we can route around a single failed child in an AB FatTree with two additional hops (three hops in total, but one replacing the link that would have been traversed anyway), without any pre-computation or coordination.

Suppose, without loss of generality, that the failed child (v) is located in a type A subtree. By construction, the parent (u) has connections to $p/2-1$ children in type A subtrees, and $p/2$ children in type B subtrees. Each of these children has $p-1$ other parents (u's siblings), which all have a link into the affected subtree. By detouring through one of its siblings, u can establish a path.

Not any sibling will work. With only local information, u must assume that the entire switch v has failed, rather than just the link $\langle u,v\rangle$. If so, none of the other parents of v have a route to the affected subtree. We call this set of v's parents a *failure group* and identify it by a tuple (t,j) consisting of v's *subtree type* t and its *index* j, since each parent is connected to the jth node in all type t subtrees. In this example, we would denote the failure group of v as $(\mathrm{A},0)$. Figure 3 shows $(\mathrm{A},0)$ and $(\mathrm{B},2)$ failure groups.

All of u's children in type A subtrees only have parents in the $(\mathrm{A},0)$ failure group, and thus cannot reach the target prefix. Thus, in Scheme 1, u will simply pick a random child, say x, in a type B subtree. By construction, x has parents in all type A failure groups, and thus *any parent of x except u does not route through v*. One of the alternate paths from u to v's subtree is shown by the bold, green line in Figure 3. This does not exist in a standard FatTree.

Multiple failures can be handled in most cases. When failures are located on different levels of the tree, Scheme 1 will always find a path. Multiple failures on the same level can sometimes block Scheme 1. For the first hop, u has $p/2$ links into type B subtrees; if none of these links work ($p/2+1$ targeted failures) then u must use Scheme 2. At the second hop, if x has no other working parents (p targeted failures and a $p/2$ random choice) then the scheme fails and packets will be dropped for the brief period until the pushback mechanism (described in Section 5.2) removes u from all such paths. At the third hop, if the link from u' into the affected subtree has also failed (2 targeted failures and $(p/2)(p-1)$ random choice), u' will invoke local rerouting recursively.

Scheme 2 – five-hop rerouting. We saw that in some cases of at least $p/2+1$ failures, Scheme 1 will fail because u will have no working links to type B subtrees. This situation trivially arises in the case of any single failure in a standard FatTree, so our work can also be seen as showing how to do local rerouting in a standard FatTree. Scheme 2 uses u's type A children, but it must go two levels down to find a working route to v, for a total of four additional hops in the detour path. One such path is illustrated in Figure 3 using the bold, dashed blue line. In Scheme 2, u picks any type A child $y \neq v$ in a different type A subtree, y picks any of its children, and that child proceeds to use normal routing to v's prefix after ensuring it routes through a parent (y's sibling) not in a currently-known failure group. This results in a five-hop path from u to the target prefix. Scheme 2 can fail in the presence of sufficiently many (at least p) targeted failures and unlucky random choices. These unlikely cases will be resolved by our pushback schemes, described next. *With fewer than p failures, local rerouting will always succeed.*

5.2 Pushback Flow Redirection

The purpose of local rerouting is to find a quick way to reestablish routing immediately after detecting a failure. The detour paths it sets up are necessarily inflated, and the schemes we use can sometimes fail although a working path exists. We introduce pushback redirection to reestablish direct routes and handle cases where local rerouting fails, but where connectivity is still possible. Pushback solves both of these issues by sending a failure notification back along each affected path to the closest switch that has a direct route that does not include the failure. The AB FatTree enables notifications to occur closer to the failure than in a regular FatTree. Reducing notifications speeds recovery and minimizes network state.

Consider Figure 4, which shows an AB FatTree built with 6-port switches. This figure shows pushback propagation in the network when the link $\langle u,v\rangle$ has failed. A total of 14 pushback messages are sent (indicated by the bold red lines), and state has to be installed at the 8 switches marked with red circles. Note that in our pushback scheme all messages indicate link failures, not node failures. If the entire node v had failed, u's two siblings would also send pushback messages along the red dashed lines, for a total of 32 additional messages and an additional 12 switches installing state. The main difference between AB FatTrees and standard FatTrees is that AB FatTrees can install state higher in the tree, and as a result, pushback messages travel less far, meaning that the network will find direct paths more quickly with less effort.

When to push back. There are three key scenarios in which a switch u will push notifications to its neighbors (we outline other uses for pushback in handling data center congestion in Section 6):

- u cannot route to some prefix in its subtree, either because of the failure of an immediate child v or upon receiving a notification from v of a failure further downstream. Then u will multicast that it can no longer route to the affected prefix to all of its neighbors excluding v.
- When all uplinks from u have failed, u will inform its children that they should use other routes.

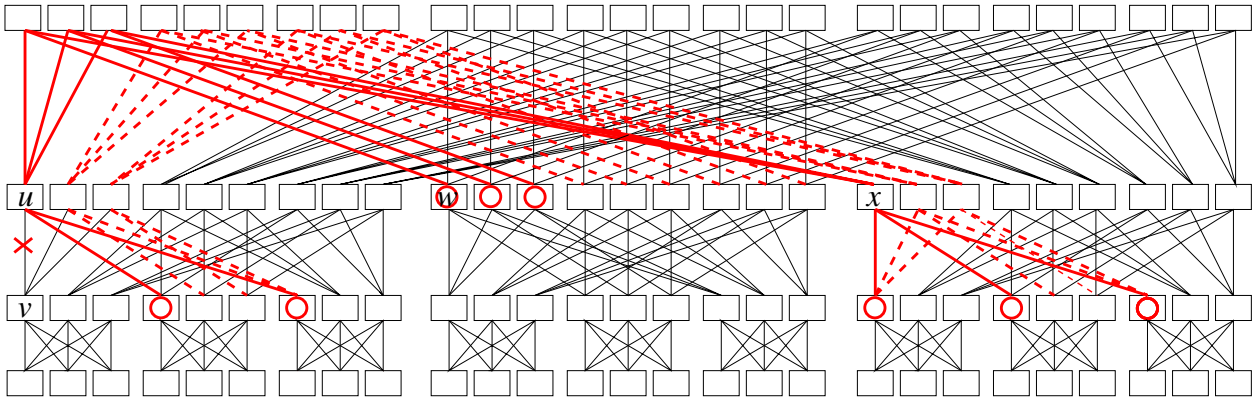


Figure 4: Illustration of pushback when the link from u to v fails (marked by the red 'X'). Solid red lines are the paths along which the notification travels, and the switches with red circles are the set of nodes that need to be notified of the failure. In the case of the entire switch v failing, the dashed red lines show the paths along which associated notifications travel and state would be installed at all the endpoints they touch.

- There may exist some external prefixes for which u is unable to route traffic if all uplinks are failed or affected by failures and the affected prefixes happen to overlap. u will inform its children so they can reroute.

Implementation sketch. We do not have the space in this paper to fully lay out the implementation details for our pushback protocol. Instead, we present here the messages, data structures, and basic mechanisms used by F10.

To handle the three pushback scenarios, laid out above, we define two types of pushback messages. **PBOnly** messages indicate that the sender cannot route to the specific prefix indicated in the message. **PBExcept** messages mean that the sender cannot reach any prefix except its own subtree (or the subtree indicated in the message). Together, **PBOnly** and **PBExcept** can represent any set of routable prefixes. **PBOnly** messages are used in scenario 1 described above, **PBExcept** messages match scenario 2, and a combination of both is used in scenario 3.

Suppose a node n receives a **PBOnly** message telling it that the edge $\langle u, v\rangle$ has failed. How does it know whether it can route around the failure—in which case it installs pushback state locally and does not forward the message—or whether it needs to forward the notification on to its neighbors? The intuition behind this is that if a node n can connect to a root node that node u cannot (in the absence of failures), then n has paths using this root that can reach v's subtree without going through the failed edge. Thus when the edge $\langle u, v\rangle$ fails, n has an alternative path to v's prefix if and only if it is connected to such a root.

One simple way that n is guaranteed to be wired to a root that u is not: when n is located at a lower level than u, then at most one of its parents routes through u, and an alternative path exists. In AB FatTrees, pushback state can be sometimes be stored higher in the tree.

To implement a method by which n at a level above u can know that it has an alternative root, we use a *subtree type stack* that represents the types of the trees on the path from a given switch to the roots of AB FatTree. When a switch that receives a pushback notification has the same type stack as the originator (or partial type stack, if the recipient is higher in the tree), then the switch has no alternative route and must forward the message on to its neighbors. In Figure 4, u and w both have stacks $\{\mathrm{A}\}$, while x has a type stack $\{B\}$. Since u and w have the same type stack, when v fails neither u nor w can route around it. x can reroute as long as it uses a parent it does not share with u and w. *Formally, a node in a subtree has a path around a single failed edge precisely if* (i) *it is at a lower level than the failure, or* (ii) *its subtree type stack is different than the top of the type stack of the failure.*

The above procedure describes how a switch u would handle receiving a notification of a single edge failure. More generally, its currently-installed pushback state tells u what prefixes its uplink is unable to serve. Any new failure (either via a failed link, or notification of a failure from an uplink) can potentially imply a prefix to which u can no longer route. If that is the case, u propagates a notification to all its downlinks.

5.3 Epoch-based Rerouting

After pushback terminates, all traffic will be routed along shortest paths (provided a route exists), but load may be unbalanced. Traffic that would have traversed failed links are shunted onto the remaining links. The third step is then to repair load balancing by reassigning flows. This is a global process that is somewhat more involved than the previous two schemes, so while failures are immediately reported to a centralized controller, the rebalancing of load occurs periodically at discrete epochs.

We describe a centralized load balancing server in Section 6.2; the same mechanism is used to rebalance flows after failures. The mechanism for reporting traffic characteristics and scheduling will be discussed subsequently.

Failures are communicated to the centralized controller and taken into account in scheduling. Only shortest paths are considered by the controller—local detours are intended to be a temporary patch. Since all paths have the same length, the controller assigns flows to minimize the maximum traffic across any link. If there is no direct path available, the flow will continue to take a locally rerouted path if possible. Additionally, if a packet from a scheduled flow encounters a failed link or node before the centralized controller is informed or reflects the change, it is treated as non-scheduled from that point onwards. If it remains stable, it will be rescheduled in the next epoch.

When a node recovers, the switch or link must prove that it is stable by remaining up for an extended period of time before the centralized scheduler will assign it traffic. This minimizes lost packets due to repeated failures of flaky devices. By putting recovery of hardware on a somewhat slower time scale, we aggregate frequent and correlated failures into a single event and only incur the compulsory losses once. When the controller does decide to reinstall the device, all neighbors are informed, and they are responsible for tearing down local reroutes and pushback blocks. Only when the neighboring switches acknowledge reinstallation is complete does the central controller use the new device for scheduled flows.

5.4 Failure Detection

Most current detection methods intentionally ensure that devices do not react to failures too quickly [25]. In IP routers, OSPF and IS-IS, by default, implement 330 millisecond heartbeats with 1 second dead intervals. Similarly, layer 2 Ethernet switches will report failures only after a waiting period on the order of multiple milliseconds. (This is called debouncing the interface.) Most of these failure detection methods only declare a failure after multiple, relatively slow heartbeats because the networks they traditionally handle are not necessarily physically connected and/or operate on shared media. In these settings, congestion can cause false positives, and routing algorithms are prone to instabilities during rapid changes.

To achieve near-instantaneous rerouting, we need to be able to rapidly and accurately detect failures. In the case of fail-stop behavior, we need a faster failure detector that does not depend on multiple losses of relatively infrequent heartbeats as *mean time to recovery* is bounded by the time to detection, plus the time to compute and install any changes into the routing table. In the case of stochastic failures, we need a more accurate failure detector that does not rely on the loss of a few designated packets.

F10 is able to achieve fast neighbor-to-neighbor failure detection because switches are directly connected and routing loops are impossible by construction. Our failure detection mechanism requires that switches continually send packets, even when idle. These packets test the interface, data link, and to an extent, the forwarding engine. F10's failure detector takes advantage of the fact that packets should be continually arriving, and allows the network administrator to define two sets of values—one for bit transitions to detect physical layer issues and one for valid packets and forwarding logic to detect link- and network-layer problems (higher-level failures require higher-level, potentially end-to-end solutions):

- t, the time period over which to aggregate
- c, the required number of bit transitions/valid packets per t for a working link to not be declared as down
- d, the number of bit transitions/valid packets per t before a failed link is brought back up

These values allow for the customization of the threshold for stochastic losses, as well as the amount of time necessary before the link can be declared as down. There are several factors that a operator can take into account when choosing appropriate values for these variables:

- Probability distributions of failure and recovery times
- Desired false positive rate
- Application requirements for reactivity to failures

While there is a fundamental trade-off between stability and reactivity (more aggressive c/t ratios necessitate a higher aggregation period), t is ideally set to allow for recovery before a transport-layer timeout. Note that these values will not result in persistent flapping because we use exponential backoff to handle fluctuations.

Our system eliminates the usual concerns with fast failure detection. Firstly, our failover protocols only deal with one link at a time, meaning that a spurious failure will not affect any other link, cascade failures, or create feedback loops. The only negative consequence of a spurious failure is that the increased load from reroutes will cause congestion. However, local rerouting is intended to be short-term. Further, global load balancing is done based on the measured end-to-end traffic matrix, ignoring the temporary detour routes.

Secondly, local rerouting is initiated and can be removed at the affected node. Instead of having an extended period during which the network propagates status updates until the system converges, our rerouting protocol completes in the time it takes for a switch to update its routing table. The choice of whether to send along the link in question or to deflect to a new path is made at the detecting switch, thus limiting the issue of convergence of local rerouting to a single switch and guaranteeing that the protocol converges before the next failure.

Note that continuous probing assumes certain properties of the link layer—particularly that it is full-duplex. The type of Ethernet used in data centers are mostly full-duplex between switches, and in fact, Cisco gigabit Ethernet switches and Ethernet standards starting from

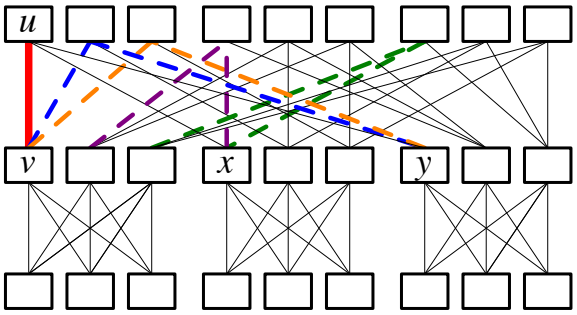


Figure 5: Example where the bold, red link between u and v becomes congested. If the congestion is in the downward direction, at least x and y need to be notified of the congestion to alleviate it. The colored, dashed lines indicate the four paths toward which traffic will be shifted as a result of said notification.

10GbE do not even support half-duplex or CSMA/CD. Furthermore, we argue that this functionality is practical since modern Ethernet standards already call for continual broadcasting of null symbols during idle periods.

6 Load Balancing

A closely related problem to failure recovery is that of load balancing. Not only do failures increase load on the rest of the system, they also have very similar characteristics. Traffic in data centers, like failures, follows a long-tailed distribution [5]. The majority of flows are small and short-lived, but their longer-lived counterparts can cause long-term congestion if not handled correctly.

Even in the case of a single loaded link, the two problems share much in common. Consider Figure 5, where a single link connecting u and v is overloaded. v can detect and instantly react if the load is in the (v,u) direction, but if it is in the reverse direction, the closest nodes that can respond to the issue are x and y. In the end, however, *all* links in the network could potentially need to change in order to restore global load balancing. We take the same 'cascading' approach to load as we do with failures and introduce three mechanisms that mirror those above:

- A flow-placement mechanism that allows each switch to locally place flows based on expected load.
- A version of our pushback mechanism that is able to gracefully handle momentary spikes in traffic. For details, we refer the interested reader to our tech report [19].
- The same epoch-based centralized scheduler that is also used for failure recovery.

Because TCP dynamics make packet reordering undesirable, we place traffic on a per-flow basis. At a high level, the centralized scheduler preallocates a portion of each link for long-term, stable flows. The remainder is used for new and unstable flows—these are randomly scheduled in the remaining capacity, but with pushback to deal with short term congestion.

6.1 Weighted Random Load Balancing

Traffic that is too short-lived to benefit from our centralized scheduling algorithm needs to be handled locally and immediately. For these types of flows, switches on the upward path use random placement of short-term traffic across all of the available shortest paths. Each flow is directed along upward edges randomly, and in the case that the centralized scheduler makes paths unequal in terms of scheduled load, we use weighted ECMP that is based on the residual capacity left after scheduling.

Note that new links have an initial residual capacity of zero, and thus, new flows do not use the link so that the centralized controller is able to ensure consistent weighting. If all links have zero remaining capacity, a new flow is placed across some non-failed link with equal probability.

In the example in Figure 5, random load balancing attempts to avoid congesting the link in the first place by distributing across all available links. After the congestion occurs, v will place less weight on the congested link, and even x and y will need to adjust their weights after pushback load balancing completes.

When there are no failures and stable flows, just placing flows randomly across all paths can achieve optimality. However, spontaneous congestion and failures necessitate other mechanisms in addition to random load balancing—mechanisms like pushback and centralized scheduling.

6.2 Centralized scheduling

Longer-term, predictable flows can and should be scheduled centrally to ensure good placement to avoid persistent congestion. For these longer flows, we use a similar approach to MicroTE [6], which advocates centralized scheduling of ToR-to-ToR pairs that remain predictable for a sufficient timespan. The authors found from measurement data that data center traffic is predictable. They propose a system in which a server in each rack saves traffic statistics and periodically sends to a centralized controller a list of "predictable" flows that have instantaneous values within some delta of their average value.

In F10, these flows are scheduled with a greedy algorithm that sorts the flows from largest to smallest and places them in order on the paths with the least cost, where the cost of a path is defined as $\sum \frac{1}{R(e)}$ over the edges e in the path P, where $R(e)$ is the remaining capacity of edge e. The controller informs ToRs about scheduled flows, and residual capacities are sent to each switch to use for weighted ECMP. If a scheduled flow runs into a failure, it becomes unscheduled at the point of failure, and gets placed using weighted ECMP.

In general, optimal rearrangement is an NP-complete problem for single-source unsplittable flows. We choose the greedy algorithm for scalability reasons, but the exact choice of algorithm is orthogonal to our work. Multipath

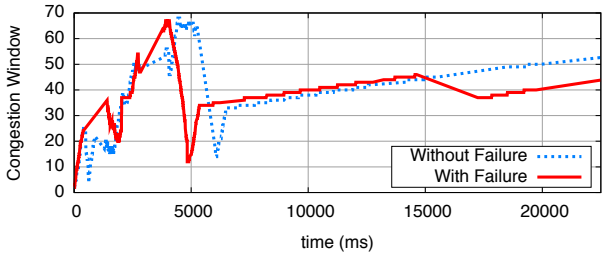


Figure 6: TCP congestion window trace with and without failure. In the case of the failure, a link went down at 15sec and F10 recovered before a timeout occurred.

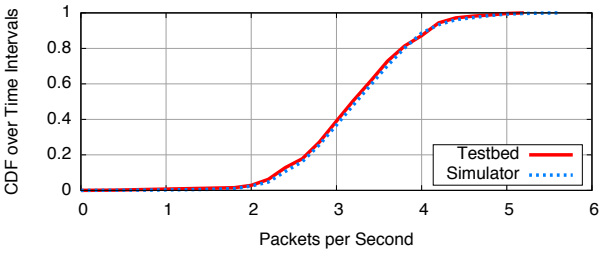


Figure 7: Comparison of throughput of the testbed and the simulator through ten failures and the same topology/offered load.

flows are more flexible from a load balancing perspective, but require end host changes to the TCP stack.

7 Prototype and Evaluation

7.1 Prototype

We built a Click-based implementation of F10 and tested it on a small deployment in Emulab [24]. The prototype runs either in user-mode or as a kernel module. The implementation is a proof of concept and correctly performs all of the routing and rerouting functionality of F10. It is able to accept traffic from unmodified servers and route them to their correct destinations.

Failure Characteristics. We instrumented a Linux kernel to gather detailed TCP information, including accurate information about congestion window size; we used this instrumented kernel to test the effect of a failure on a TCP stream. Tests were performed in Emulab, but since bandwidth limitations in both the links and the Click implementation are lower than in a real data center, we lowered the packet size so that the transmission time and the number of packets in flight are comparable to a real deployment with 1 gigabit links. We used this testbed to compare the evolution of a congestion window with and without failure during a 25 second interval in Figure 6. F10 is able to recover from the failure before a timeout occurs and the performance hit is minimal.

Failure Detector. We have also implemented an approximation of F10's failure detector using Click in polling mode. The detector would ideally be built in hardware, but preliminary results indicate that we can approximate the ideal detector with a Click-based implementation. Unfortunately, with Click, it is not possible to track bit transitions on the wire, and there is some amount of jitter between successive schedulings of the network device poller. Even so, our Pentium III testbed machine was able to accurately detect failures after as little as 30μs—much less than a single RTT in a data center. With this property, we were able to fail based on the rate of valid packets.

At each output port, we placed a strict priority scheduler that pulls from the output queue if possible, or else generates a test packet. The dummy packets are intercepted and dropped by the downstream failure detector before being passed to the rest of the system. The detector asserts a failure and notifies the rest of the system when the arrival rate of either good or nonce packets drops below the specified threshold.

7.2 Evaluation Environment

Simulator. We created an event-driven simulator to test the efficacy of F10 with medium- to large-scale data centers—resources limited the feasibility of such experiments in our testbed setting. The simulation includes the entire routing and load balancing protocol along with the fast failure detection algorithm.

Our multicore, packet-level, event-driven simulator comprises 4181 lines of Java. It implements both low-level device behaviors and protocols. The Layer 2 Ethernet switches use standard drop-tail queues and have unbounded routing state; our evaluation shows that even with many failures in the network, only a modest amount of state needs to be installed. The simulator models 100 ns latency across each link to cover switch and interface processing as well as network propagation latencies. When there is no traffic, each switch generates nonce messages to its neighbors. The link is marked as failed if three consecutive packets are not received correctly.

Our experiments are performed assuming 24-port 10GbE switches in a configuration that has 1,728 end hosts, resulting in a standard or AB FatTree with three layers. Except in Section 7.6, we use UDP traffic in our experiments so that we can more precisely measure the impact of the failure on load. This enables us to understand how well the evaluated mechanisms improve *network capacity*. TCP will generally back off quickly, resulting in lower delivered throughput than shown here.

We have compared the measurements generated by both the testbed and simulator, for an identical topology and offered load. Figure 7 is a CDF of throughput for a single source-destination pair that experienced a sequence of ten failures, which each went through all of the stages of failover in F10. We found that, in all cases tested, the simulator and testbed results matched each other closely.

Workload model. We derive our workload from measurements of Microsoft data centers given by Benson et al. [5]. We generate log-normal distributions for (1)

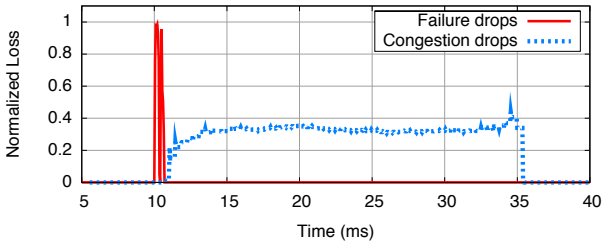


Figure 8: Aggregate losses due to lack of connectivity and congestion in the case a single failure.

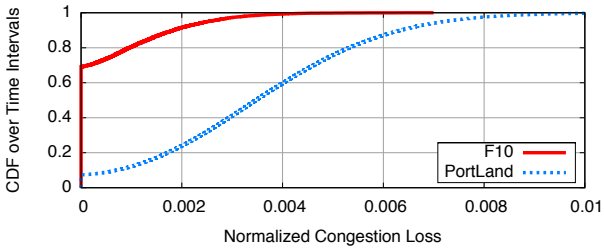


Figure 9: CDF of the congestion losses of both PortLand and F10 under realistic traffic and failure conditions.

packet interarrival times, (2) flow ON-periods, and (3) flow OFF-periods, parameterized to match the experimental data from the paper. In certain experiments (labeled explicitly below), we scale the packet interarrival times to adjust the load on the network.

Failure model. Failures are based on the study by Gill et al. [10] that investigated failures in modern data centers. We generated log-normal distributions for (1) the time between failures and (2) the time to repair for both switches and individual links based on their experimental data.

Note that we do not consider leaf (ToR) switch failures, as these are well handled by cloud software. Fault tolerance of rack failures is orthogonal to our work on the robust interconnection between them.

7.3 Recovering from a Single Failure

Figure 8 shows a breakdown of the losses over time after a single switch failure in F10 running a uniform all-pairs workload at 50% (UDP) load. The *y*-axis in this graph shows the loss rate normalized to the expected number of packets traversing each switch.

When the failure occurs at 10ms, there is a burst of packet drops due to failure. At around 11ms, the neighbors of the failed switch detect the failure, and local rerouting installs new working routes and eliminates failure drops. Local rerouting reduces the capacity of the network, triggering congestion. When the pushback scheme is initiated later, it quickly and effectively optimizes paths, spreading the extra load and eliminating the congestion loss.

7.4 Comparison with PortLand

F10 recovered from the single failure evaluated in the prior section within 1 ms of the failure; this is more than two orders of magnitude faster than possible with PortLand [20], the state of the art research proposal for fault tolerance in data center networks, which reports minimum failure response times of 65 ms. In addition, F10 was able to recover load balancing in 35 ms, while PortLand does not handle congestion losses at all. In this section, we compare F10 against PortLand using the realistic, synthetic traffic and failure models described in Section 7.2.

Figure 9 shows the congestion rate in each system. We generated workload and failure events from a random seed and fed the same trace into PortLand, which uses a standard FatTree, and F10 with an AB FatTree and all our techniques. We aggregated loss statistics over a 500μs time interval, and report the distribution of congestion loss over these intervals. The figure aggregates data points for multiple runs that start from different initial conditions.

Overall, F10 has much less congestion than PortLand. F10 sees negligible loss for 3/4 of time periods, whereas PortLand nearly always has congestion. In total, Portland has 7.6$\times$ the congestion loss of F10 for UDP traffic.

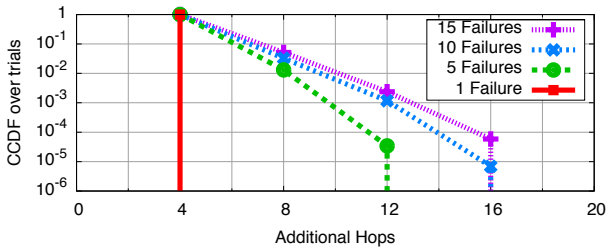
7.5 Local Rerouting and AB FatTrees

Note that both standard and AB FatTrees can perform local rerouting, but the former is unable to exploit the shorter detours of F10. Here, we evaluate the impact of the novel structure of AB FatTrees during local reroutes. We measured the path inflation of local reroutes using varying numbers of switch failures (up to 15 concurrent failures, implying up to 360 failed links) in standard vs AB FatTrees (see Figure 10). We found that *local reroutes in AB FatTrees experience roughly half the path inflation than in standard FatTrees*, owing to F10's ability to use Scheme 1 rerouting in addition to Scheme 2. Even for many concurrent failures, the vast majority—more than 99.9%—of reroutes use the minimum number of hops (2 for AB FatTrees). We also looked at random link failures as opposed to switch failures, and obtained similar results in terms of how the path dilation in F10 compares with that of standard FatTrees.

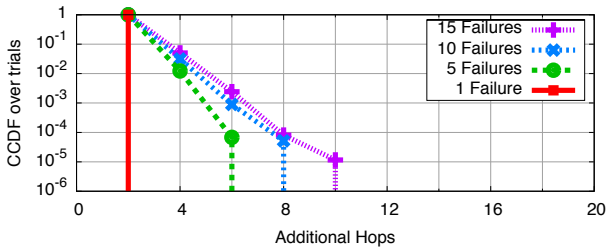
7.6 Speeding up MapReduce

We conclude our evaluation by simulating the behavior of a MapReduce job (with TCP flows) in our data center. We used a MapReduce trace generated from a 3600-node production data center [7], and considered the performance of just the shuffle phase, where flows are initiated from mappers to reducers, with mappers and reducers assigned randomly to servers. We focus our study on only those MapReduce computations that involved fewer than 200 mappers and reducers in total.

Figure 11 compares the performance of the shuffle operation under the two architectures—F10 and PortLand—and the failure model used thus far. Since the shuffle operation completes only after all the constituent flows

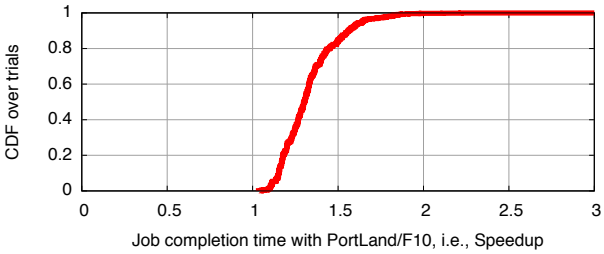


(a) A standard FatTree, which can only use local rerouting Scheme 2

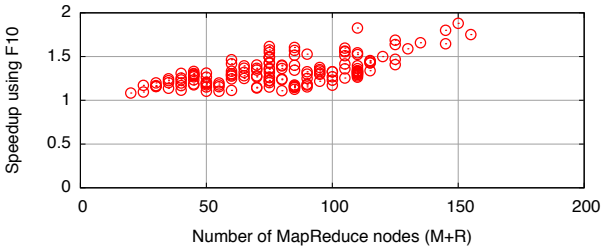


(b) An AB FatTree using both local rerouting schemes

Figure 10: Complementary CDF of the path dilation using local rerouting for 1, 5, 10 and 15 simulated switch failures when using our local rerouting schemes in standard and AB FatTrees.



(a) Distribution of Speedup



(b) Speedup vs Job Size

Figure 11: An end-to-end evaluation using PortLand or F10 for MapReduce jobs.

are complete, it suffers from the well-known stragglers problem. If any of the flows traverse a failed or rerouted link, it suffers from suboptimal performance. We measure the speedup of an individual job as the completion time under PortLand divided by that of the job under F10.

Figure 11a shows the distribution of the speedup; we find that F10 is faster than PortLand with a median speedup of about 1.3×. Figure 11b, shows the distribution of speedup vs job size, and we find that gains are larger when more nodes participate and compete for bandwidth. We conclude that F10 offers significant gains over PortLand, and this will improve in larger future data centers.

8 Extensions

8.1 Other Types of Multi-Tree Networks

So far, we have focused on a specific subset of multi-tree networks—Clos networks where the number of uplinks at any switch is equal to the number of downlinks. We now show how the ideas presented in this paper can be used in conjunction with other topologies. In particular, we generalize our protocols for any type of folded Clos network and also look at traditional data center topologies.

Oversubscribed and Overprovisioned Networks: Allowing the number of uplinks and downlinks for a single switch to differ allows for "vertical" asymmetry. Such asymmetry can be useful if different layers use different technologies (e.g., VL2's Clos topology) or when traffic patterns do not require full bisection bandwidth.

Fortunately, these networks require little to no change in our algorithms. The placement of flows by the global rebalancer is easily extended to this case. Pushback similarly does not rely on the number of links; notifications are broadcast to all uplinks and downlinks, and termination only depends on level and type stack. For basic routing, local rerouting and recursive pushback, a few generalizations of functions must be made, and for this we require configuration of the number of downlinks for switches at each level, D_{level}. All references to p should be replaced by D_{level} and protocols should be changed to take the nonuniformity into account (e.g., $prefix(a,i) = a \gg (\Sigma_{l=1}^{i}(\lceil \log(D_l) \rceil))$).

Traditional Data Center Networks: Next we look at more traditional topologies like those described in [8]. These topologies have many extra links compared to a Clos topology with an equal number of switches, but gain fault-tolerance as a result. Although we focused on next-generation data centers as they are more scalable and cost effective, there is no reason that F10's concepts cannot be applied to traditional networks as well. These networks have two main topological differences from Fat-Trees:

- Cross-links between switches in the same level
- Multiple links into a given subtree/pod (often all subroots connect to all roots and vice versa)

While these links do not necessarily add to the capacity of the network, they allow for shorter reroutes than possible in F10. Even so, F10's failure detection, near-instantaneous failover and load balancing concepts can increase performance and reactivity to failures and fault tolerance in the case of multiple failures. Note that in the case where all subroots connect to all roots and vice versa, A and B subtrees have identical wirings just as in the lowest level of normal FatTrees. Thus, any child can be used

for phase 1 of local rerouting and pushback terminates as soon as it follows a downlink.

8.2 Beyond AB FatTrees

Our architecture introduces an extra type of subtree that connects to a different set of roots and thus provides additional path diversity. A natural question to ask is whether we can get even more diversity with more types.

In the limit, we can create a p-type FatTree in which all subtrees are connected to a slightly different set of roots. This is accomplished by rotating the set of roots to which a subtree connects—subroot j of the first subtree connects to the jp through the $(j+1)p-1$ roots, subroot j of the second subtree connects to roots $jp+1$ through $(j+1)p$, and in the same manner, each additional subtree incrementally shifts by one. This guarantees that every sibling of a given node n has at least one alternative path.

At first glance, this seems to improve the potential for efficient reroutes. However, more choices at the first hop of local rerouting comes at the cost of fewer at the second. While an AB FatTree provides $p-1$ alternatives for the second hop of Scheme 1 given a single failure, a p type FatTree will have an average of $p/2-1$, with some nodes having more alternatives than others. Increasing the number of types does not, in general, increase the chance of finding a two-hop detour.

For pushback, more alternatives means that the notifications can stop earlier (in the case of a single failure in a p-type FatTree, pushback can terminate after the message traverses any downward link). However, traffic destined for the failed path is split over a smaller number of alternate paths, disproportionately increasing the load on those paths. In sum, the tradeoffs are complex, and we leave a fuller comparison for future work.

9 Related Work

The topic of fault tolerance in interconnection networks has a long history [1, 9, 16]. Most previous work on this topic, most notably [2], has added hardware in the form of stages, switches and links to existing topologies to make them more fault tolerant while keeping latency and non-blocking characteristics constant. We instead allow for a temporary increase in latency for paths affected by faults in exchange for no increase in hardware cost.

In the context of today's data centers, researchers have recently proposed several alternative interconnects. Our work directly builds on FatTrees [3] as they are used in PortLand [20], although our ideas generalize to other multi-rooted trees like VL2 [11] and beyond. We leverage many of the earlier mechanisms in our work. We replace the interconnect with our novel AB FatTree network and co-design local rerouting, pushback, and load balancing mechanisms to exploit the topology. Hedera [4] implements centralized load balancing on top of PortLand. Hedera only schedules new flows, whereas we choose to globally rearrange flows periodically.

DCell [13] and BCube [12] introduce structured networks that are not multi-rooted trees.The key difference is that these topologies trade more hardware for their increased robustness. DCell performs local rerouting after a failure but is not loop free (unlike ours). Loop freedom is important to enable fast failure detectors at the link layer without compromising reliability.

Jellyfish [23] takes a different approach to datacenter design—unstructured, random-wiring. It trades regularity and rearrangeable, non-blocking guarantees for better average-case performance with less hardware. Our mechanisms might apply to their topology, though it would require precomputation of all detour paths, and it is unclear how much path dilation would be needed on average.

Our failure recovery schemes leverage existing techniques. Our local rerouting scheme uses tags and failure lists analogous to MPLS and Failure-Carrying Packets [14], respectively. MPLS supports a similar style of immediate local detours (Fast Reroute) while waiting for the failure to propagate upstream (Facility Backup) [21]. MPLS failover requires manual preconfiguration and stored state, whereas our system has easy-to-compute backup paths and stores state only when there is a failure.

DDC [18] has the same intuition that failover should be done at the network layer. They make no assumptions about network topology, and so they cannot benefit from preset local reroutes. In order to handle unstructured networks, their approach reroutes for each destination separately and does not result in paths that are as efficient as the ones produced by our local rerouting scheme.

10 Conclusion

Scalable, cost-efficient and failure resilient data center networks are increasingly important for cloud-based services. In this paper, we describe F10, a novel multi-tree topology and routing algorithm to achieve near-instantaneous restoration of connectivity and load balance after a switch or link failure. Our approach operates entirely in the network with no end host modifications, and experiments show that routes can generally be reestablished with detours of two additional hops and no global coordination, even during multiple failures. We couple this fast rerouting with complementary mechanisms to quickly reestablish direct routes and global load balancing. Our evaluation shows significant reduction in packet loss and improved application-level performance.

Acknowledgments

We gratefully acknowledge our shepherd George Porter for guiding us through the shepherding process. This research was partially supported by the National Science Foundation grants CNS-0963754 and CNS-1040663.

References

[1] G. B. Adams, III, D. P. Agrawal, and H. J. Seigel. A survey and comparision of fault-tolerant multistage interconnection networks. *Computer*, 20:14–27, June 1987.

[2] I. Adams, G.B. and H. Siegel. The extra stage cube: A fault-tolerant interconnection network for supersystems. *IEEE Trans. Comput.*, C-31(5):443–454, May 1982.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[4] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: dynamic flow scheduling for data center networks. In *NSDI*, 2010.

[5] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[6] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine grained traffic engineering for data centers. In *CoNEXT*, 2011.

[7] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, 2011.

[8] Cisco. Data center: Load balancing data center services. `https://learningnetwork.cisco.com/servlet/JiveServlet/downloadBody/3438-102-1-9467/cdccont_0900aecd800eb95a.pdf`.

[9] C. C. Fan and J. Bruck. Tolerating multiple faults in multistage interconnection networks with minimal extra stages. *IEEE Trans. Comput.*, 49:998–1004, September 2000.

[10] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *SIGCOMM*, 2011.

[11] 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.

[12] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: a high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, 2009.

[13] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: a scalable and fault-tolerant network structure for data centers. In *SIGCOMM*, 2008.

[14] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica. Achieving convergence-free routing using failure-carrying packets. In *SIGCOMM*, 2007.

[15] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann Publishers, Inc., 1992.

[16] F. T. Leighton and B. M. Maggs. Fast algorithms for routing around faults in multibutterflies and randomly-wired splitter networks. *IEEE Trans. Comput.*, 41:578–587, 1992.

[17] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34:892–901, October 1985.

[18] J. Liu, B. Yang, S. Shenker, and M. Shapira. Data-driven network connectivity. In *HotNets*, 2011.

[19] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A fault-tolerant engineered network. UW-CSE-12-09-05, 2012.

[20] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: a scalable fault-tolerant Layer 2 data center network fabric. In *SIGCOMM*, 2009.

[21] P. Pan, G. Swallow, and A. Atlas. Fast reroute extensions to RSVP-TE for LSP tunnels. *Internet RFC 4090*, 2005.

[22] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath TCP. In *SIGCOMM*, 2011.

[23] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: networking data centers randomly. In *NSDI*, 2012.

[24] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, 2002.

[25] R. White. High availability in routing. `http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_7-1/high_availability_routing.html`.

LOUP: The Principles and Practice of Intra-Domain Route Dissemination

Nikola Gvozdiev, Brad Karp, Mark Handley
University College London

Abstract

Under misconfiguration or topology changes, iBGP with route reflectors exhibits a variety of ills, including routing instability, transient loops, and routing failures. In this paper, we consider the intra-domain route dissemination problem from first principles, and show that these pathologies are not fundamental–rather, they are artifacts of iBGP. We propose the Simple Ordered Update Protocol (SOUP) and Link-Ordered Update Protocol (LOUP), clean-slate dissemination protocols for external routes that do not create transient loops, make stable route choices in the presence of failures, and achieve policy-compliant routing without any configuration. We prove SOUP cannot loop, and demonstrate both protocols' scalability and correctness in simulation and through measurements of a Quagga-based implementation.

1 Introduction

Much has been written about iBGP's susceptibility to persistent routing instability and forwarding loops [11, 12, 20]. Yet the transient behavior of intra-domain dissemination of external routes has been, to our knowledge, unexamined. In recent work, we found that today's iBGP frequently incurs transient forwarding loops while propagating updates [13]. Real-time traffic of the sort prevalent on today's Internet does not tolerate transient loops or failures well; Kushman *et al.* [18] note that periods of poor quality in VoIP calls correlate closely with BGP routing changes. Even a BGP-free core does not entirely eliminate iBGP's role in intra-AS route dissemination, nor any associated pathologies: border routers (BRs) must still use iBGP internally to exchange routes.

In this paper, we illustrate that the routing instability and transient loops that often occur when disseminating a route update (or withdrawal) learned via eBGP within an AS are not fundamental. Rather, they are side-effects of the way in which route dissemination protocols—not only iBGP as typically deployed with Route Reflectors (RRs), but in the case of transient loops, alternatives such as BST and RCP, as well—happen to disseminate routes. It is the lack of attention to the *order* in which routers adopt routes that causes these pathologies.

Based on these observations, we introduce the Simple Ordered Update Protocol (SOUP), a route dissemination protocol that *provably* never causes transient forwarding loops while propagating any sequence of updates from any set of BRs. SOUP avoids causing loops by reliably disseminating updates hop-by-hop along the *reverse forwarding tree* from a BR. We further introduce the Link-Ordered Update Protocol (LOUP), which uses a similar ordering mechanism to avoid loops, but includes optimizations that reduce its convergence time (compared with that of SOUP) in common cases.

We are not the first to observe that careful attention to the details of route propagation can eliminate transient anomalies. DUAL [8], the loop-free distance-vector interior gateway protocol (IGP), explicitly validates before switching to a next hop that doing so will not cause a forwarding loop. And Consensus Routing [17] augments eBGP with Paxos agreement to ensure that all ASes have applied an update for a prefix before any AS deems a route based on that update to be stable. SOUP and LOUP use comparatively light-weight mechanisms (*i.e.,* forwarding in order along a tree) to avoid transients during route dissemination within an AS.

Our contributions in this paper include:

- a first-principles exploration of the dynamics of route dissemination, including how known protocols do dissemination and the trade-offs they make
- invariants that, when maintained during route dissemination, avoid transient loops when any set of updates to prefixes is introduced
- SOUP, a simple route dissemination protocol that enforces these invariants using ordered dissemination of log entries along a tree
- a proof that SOUP cannot introduce forwarding loops
- LOUP, an optimized route dissemination protocol that converges faster than SOUP
- an evaluation in simulation that demonstrates the correctness and scalability of LOUP on a realistic Internet Service Provider topology
- measurements of an implementation of LOUP for Quagga that show LOUP scales well in memory and computation, so that it can handle a full Internet routing table and high update processing rate.

2 Intra-AS Route Dissemination

Internet routing consists of three components: *External BGP* (eBGP) distributes routes between routing domains and is the instrument of policy routing. An *Interior Gateway Protocol* (IGP) such as OSPF or IS-IS tracks reachability within a routing domain. Finally, *Internal BGP* (iBGP) distributes external routes received by border routers (BRs) both to all other BRs, so they can be redistributed to other domains, and also to all participating internal routers. In this paper we are concerned with iBGP: when a route changes elsewhere in the Internet, how does this change propagate across a routing domain?

When a BR receives a route change from a neighboring routing domain or *Autonomous System* (AS), it sanity checks it and applies a policy filter. This policy filter can drop the route, or it can modify it.

After policy filtering, the BR runs its decision process, determining whether it prefers this route to other routes it may hold for the same IP address prefix. The decision process is specified in the BGP standard, and consists of successive rounds of eliminating candidate routes based on different criteria until only one remains. First in the decision process is Local Preference, so configured policy trumps all else. Lower down the list come AS Path length, and below that IGP distance (the distance to the BGP next hop—usually either the announcing BR itself, or its immediate neighbor in the neighboring AS).

When a BR receives a route announcement for a new prefix, if it is not dropped by policy, the BR distributes it to all other routers in the domain so they can reach this destination. If a BR already has a route to that prefix, it only sends the new route to the other routers if it prefers the new route. Similarly, if a BR hears a route from another BR that it prefers to one it previously announced, it will withdraw the previously announced route.

Having decided to announce or withdraw a route, it is important to communicate the change reliably and quickly to the rest of the AS. BGP routing tables are large—currently over 400,000 prefixes—and multiple BRs can receive different versions of each route. Periodic announcement of routes doesn't scale well, so dissemination needs to be reliable: once a router has been told a route, it will hold it until it is withdrawn or superseded. Route dissemination also needs to be fast, otherwise inter-AS routing can take a long time to converge.

The simplest way to disseminate routes across an AS is full-mesh iBGP, where each router opens a connection to every other router in the domain (Fig. 1a). When an update needs to be distributed, a BR just sends it down all its connections. TCP then provides reliable in-order delivery of all updates to each router, though it provides no ordering guarantees between different recipients.

In practice, few networks run full-mesh iBGP. The $O(n^2)$ TCP connections it requires dictate that all routers in a network must be reconfigured whenever a router is added or retired, and every router must fan out each update to all $n-1$ peers causing a load spike with associated processing delays. Most ISPs use iBGP route reflectors (RRs). These introduce hierarchy; they force propagation to happen over a tree[1] (Fig. 1b). Updates are sent by a BR to its reflector, which forwards them to its other clients and to other reflectors. Each other reflector forwards on to its own clients.

Route reflectors significantly improve iBGP's scaling, but they bring a range of problems all their own. In particular, each BR now only sees the routes it receives directly via eBGP and those it receives from its route reflector. Thus no router has a complete overview of all the choices available, and this can lead to a range of pathologies, including persistent route oscillations [9].

ISPs attempt to avoid such problems by manually placing route reflectors according to guidelines that say "follow the physical topology"; not doing so can cause suboptimal routing [20]. Despite these issues, almost all ISPs use route reflectors and, with conservative network designs, most succeed in avoiding the potential pitfalls.

Persistent Route Oscillations With eBGP, inconsistent policies between ISPs can lead to persistent BGP oscillations. These can be avoided if BGP policies obey normal autonomous systems relations ("obey AR" [7]). Essentially this involves preferring customer routes to peer or provider routes, and that the graph of customer/provider relationships is acyclic. However, even when AR is obeyed, BGP's MED attribute can result in persistent iBGP route oscillations [10].

Briefly, MED allows an operator some measure of control over which link traffic from his provider takes to enter his network. Unfortunately the use of MED means that there is no unique lexical ordering to alternative routes. The decision process essentially takes two rounds; in the first routes received from the same ISP are compared, and the ones with higher MED are eliminated; in the second, the remaining routes are compared, and an overall winner is chosen. Thus route A can eliminate route B in the first round, then lose in the second round to route C. However, in the absence of route A, route B may win the second round. Compared pairwise, we have $A > B$, $B > C$, and $C > A$. To make the correct decision, routers must see all the routes, not a subset of them.

Route reflectors hide information; only the best route is passed on. This interacts poorly with MED, resulting in persistent oscillations [19]. Griffin and Wilfong prove that so long as policies obey AR, full-mesh iBGP will always converge to a stable solution [10]. The same is not true of route reflectors or confederations. To avoid iBGP route oscillations, it is sufficient to converge to the same routing solution that would be achieved by full-mesh iBGP, and we adopt this as a goal.

The Rise of the BGP-free Core In recent years many ISPs have deployed MPLS within their networks, primarily to support the provisioning of VPN services. MPLS also allows some networks to operate a BGP-free core.

An MPLS network with a BGP-free core functions in the same way as BGP with route reflectors, except that only BRs are clients of the RRs. An internal router that only provides transit services between BRs does not need

[1] ISPs often use two overlaid trees for redundancy.

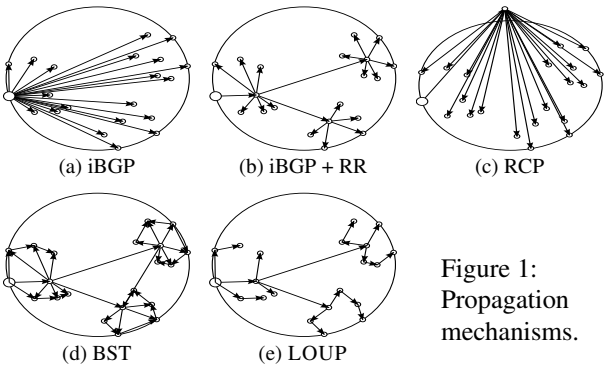


Figure 1: Propagation mechanisms.

to run BGP; instead the entry BR uses an MPLS label-switched path to tunnel traffic to the exit BR. A protocol such as LDP [2] is then used to set up a mesh of MPLS paths from each entry BR to each exit BR.

One potential advantage of a BGP-free core is that core routers need only maintain IGP routes, rather than the 400,000 or so prefixes in a full routing table, though they must also hold MPLS state for a subset of the $O(n^2)$ label-switched paths. In general though, due to improvements in hardware and forwarding algorithms [23], the overall size of routing tables is not the problem it was once thought to be [6]. Another potential advantage of a BGP-free core is that it reduces the number of clients of iBGP route reflectors. Fewer clients mean less processing load on the RRs and less chance of configuration errors that cause routing instability.

Against these potential benefits, a BGP-free core depends on additional protocols such as LDP or RSVP-TE for basic connectivity, which add complexity and themselves need careful configuration. Moreover, a BGP-free core doesn't eliminate transient forwarding loops. BRs still run iBGP with RRs, and RRs still fail to control the order in which their distinct clients hear updates. Although MPLS may reduce the prevalence of such loops, it cannot prevent them—those transient loops that do occur will traverse the whole network between two (or more) BRs.

Thus the adoption of a BGP-free core seems to be driven by obsolete concerns about routing table size and by undesirable properties of iBGP with route reflectors. Our goal is to revisit the role played by iBGP, and demonstrate that iBGP's limitations are not fundamental. We will describe replacements for iBGP that:

- are not susceptible to configuration errors,
- are stable under all configurations,
- are not prone to routing protocol traffic hot spots,
- minimize forwarding table (FIB) churn, both in BRs and internal routers,
- propagate no more changes to eBGP peers than full-mesh iBGP would,
- free of transient loops.

3 Dissemination Mechanisms

We now examine alternative route dissemination mechanisms from first principles to cast light on how forwarding loops arise and inform the design of the loop-free route dissemination protocol we describe subsequently.

Although iBGP's use of TCP ensures the in-order arrival of updates at each recipient, each recipient applies every update as soon as it can. Thus the order of update application *among* recipients is arbitrary, causing transient loops and black holes until all the routers have received and processed the update. Route reflectors impose a limited ordering constraint (RR clients receive a route after their RR processes it), but except in trivial non-redundant topologies this constraint is insufficient to prevent loops and black holes.

One way to avoid both loops and persistent oscillations would be to centralize routing, as shown in Fig. 1c. When an external update arrives the BR first sends it to a routing control platform (*e.g.,* RCP [3]), which is in charge of running the decision process for the whole domain and distributing the results to all routers. As the RCP controller has full knowledge, it could be extended to avoid loops by applying updates in a carefully controlled order. However, to do so would require a synchronous update approach which, given the RTTs to each router, would be slower than distributed approaches.

In the case of IGP routing, it is well known how to build loop-free routing protocols. DUAL [8] is the basis of Cisco's EIGRP, widely deployed in enterprise networks, and uses a provably loop-free distance-vector approach. DUAL is based on several observations:

- If a metric change is received that reduces the distance to the destination, it is always safe to switch to the new shortest path. This is a property of distance-vector routing; if the neighbor sending the change is the new next hop, it must already have applied the update, and so must be closer to the destination. Therefore no loop can occur.
- If an increased distance is received, the router can safely switch to any neighbor that is closer than it previously was from the destination. This constraint is DUAL's *feasibility condition,* and these routes are known as *feasible routes.* They are safe because no matter the router's distance after the update, it still never forwards away from the destination.
- In all other circumstances, a router receiving an increased distance cannot safely make its own local decision. DUAL uses a *diffusing computation* [5] to make its choice. It *freezes* the current choice of next hop and queries its neighbors to see if they have a feasible route. If they do not, they query their neighbors, and so on, until the computation reaches a router close enough to the destination that it can make a safe

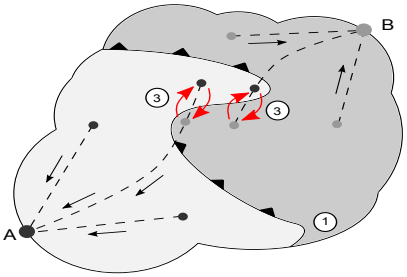


Figure 2: Transient loop when flooding an update.

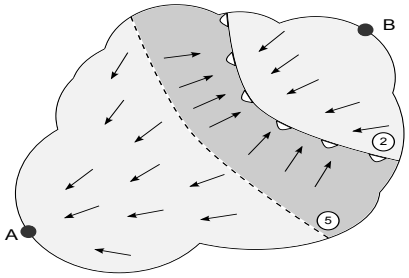


Figure 3: Withdrawal causes loop when alternate exists.

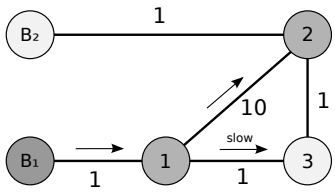


Figure 4: A simple topology exhibits looping.

choice. The responses spread out back across the network, activating routes in a wave as they return.

The iBGP route dissemination problem is different from that solved by DUAL, as iBGP distributes the same information to all internal routers, irrespective of the internal topology. DUAL, in contrast, concerns itself with path choice across the internal topology. For each prefix iBGP routers must decide between alternative external routes as those routes propagate across the network. The routes themselves do not change; rather to avoid loops we must either control the order in which route changes are received or the order in which they are applied.

Wavefront Propagation DUAL performs hop-by-hop flooding of route changes, accumulating metric changes along the way. BGP route dissemination can also be performed using hop-by-hop flooding, as shown in Fig. 1d, in which each router sends the messages it receives to all neighbors. Flooding must be done over one-hop reliable sessions to ensure messages are not lost. BST [15] takes this approach. Flooding imposes a topological ordering constraint, guaranteeing that at all times, a contiguous region of routers has processed an update. Essentially an update propagates across the domain as a *wavefront*; this is a necessary (though not sufficient) condition to avoid transient loops. iBGP does not have this property.

To see why this condition is not sufficient, even as a new route is propagated, consider Fig. 2. BR B had previously received a route to prefix P and distributed it to all the routers in the domain. BR A then receives a better route to P, and this is in the process of flooding across the domain, forming a wavefront ① flowing outward from A. All the routes in the light gray region now forward via A; the remainder still forward via B. Unfortunately, flooding does not ensure that the wavefront remains convex—that a forwarding path only crosses the wavefront once. As a result transient loops ③ can occur.

Fig. 4 shows one way that such non-convexity can occur. Initially all routers forward to some prefix via B_2, but then B_1 receives a better route. Link 1-2 would not normally be used because of its high metric. If, however, router 1 floods the update from B_1 over this link, then receiving router 2 may direct traffic towards router 3 which is on the forwarding path to B_1. As router 3 has not yet heard the update, it will direct traffic towards B_2 via router 2, forming a loop. This loop will clear eventually when router 3 hears the update. Note that the update may be delayed either by the network (*e.g.,* congestion, packet loss) or more likely, by variable update processing delays at routers.

4 Simple Ordered Update Protocol

Building upon the discussion of route dissemination primitives above, we now propose two novel dissemination techniques, *reverse forwarding tree dissemination* and *backward activation.* We combine these into a Simple Ordered Update Protocol (SOUP), and prove that it never causes transient forwarding loops within an AS.

4.1 Reverse Forwarding Tree Dissemination

Recall that BST's loops occur when one BR propagates a new route that is preferred to a pre-existing route from another BR. A sufficient condition for avoiding such loops is for *no router to adopt the new route until the next hop for that route has also adopted the route.* The condition transitively guarantees that a packet forwarded using the new state will not encounter a router still using the old state. One way to meet this condition in BGP route dissemination is for a router only to announce a route to routers that will forward via itself. Thus, route announcements flow from a BR along the reverse of the forwarding tree that packets take to reach that BR. Applying this condition in Fig. 4 precludes sending the update over link 1-2 as it is not on the RFT.

SOUP works by propagating announcements over a hop-by-hop tree, as shown in Fig. 1e. Unlike the Route Reflector tree, SOUP uses one tree per BR, rooted at that BR. SOUP builds this tree dynamically hop by hop by reversing the links on the shortest-path tree that the IGP follows to reach that BR from everywhere in the domain. This hop-by-hop nature preserves the wavefront property. Disseminating routes down the reverse forwarding tree (RFT) adds additional desirable ordering constraints that eliminate transient loops of the sort described above when improved routes are disseminated.

4.2 Sending Bad News: Backward Activation

Sending bad news is never as simple as sending good news. If a router receives a withdrawal (even over the RFT), it cannot just pass it on and locally delete the route from its FIB. If it does, a transient loop may result. Consider what happens when all routers hold more than one route to the same destination prefix. This often occurs when routes tie-break on IGP distance: more than one BR originates a route, but they are all equally preferred. Each router chooses the route to the closest exit; some choose one exit, some another. Such hot-potato routing is common when two ISPs peer in multiple places.

Withdrawing one of those routes, as *B* does in Fig. 3, causes a loop. The routers behind withdrawal wavefront ② have already switched to an alternative route via *A*. Routers farther away have not yet heard the withdrawal and still forward to *B*. Traffic loops at wavefront ②.

To avoid such loops, when a BR withdraws a route, the change must first be applied by routers furthest from the BR. Essentially we want the change to propagate in exactly the reverse of what would happen when good news propagates. In this way, no router uses the to-be-withdrawn route to forward packets to a router that has already withdrawn the route, and so the withdrawal will not cause a loop to occur. In Fig. 3 the routers farthest from *B* will remove the route first, though doing so doesn't change their forwarding decision. The routers just to the right of ② will be the first to withdraw the route and change their choice of exit router, then their parents (with respect to the tree routed at B), and so on back up to B. We call this process *backward activation*.

4.3 The SOUP Protocol

SOUP routers actively build an RFT for each BR by exchanging messages with the relevant parent router. We describe this in more detail in Section 6.1. BRs receive route updates[2] over eBGP. If a BR uses a route or the update is a change to a route it previously used, then the BR sends it hop-by-hop down the RFT.

We define a route as *active* if it is eligible to be considered in the decision process, even if it is not installed in the FIB. Updates can be sent *forward activated* or *backward activated.* Each router makes its own choice of forward or backward activation, but with one exception, once the BR has originated a route as one or the other, the update will stay that way across the network.

When a router receives an update, it checks if the route is preferred to the current activated version of the same route it received from the same BR. If it is preferred, the route is *feasible;* it is applied immediately, and previous versions of this route from the same BR are flushed. If the route wins the BGP decision process, it is installed in the FIB. Irrespective of whether it was installed in the FIB, the router then sends it on to its children as forward activated. The children will also find it to be feasible.

If the route is not feasible, the router cannot yet apply the change. Its children still have the better version of this route, and if it applies the change, it may forward to a child who forwards right back again. Instead it keeps the old route active, adds the update to a list of inactive alternative routes received from that BR, and sends the change to all its children marked as *backward activated*.

When a backward activated update is received by a leaf router on the BR's RFT, it is safe for that router to activate the update—it has no children, so no loop can result. The leaf sends an *activation message* back to its parent, indicating that it has activated the change. Once the parent receives activation messages from all of its children, it in turn can activate the update and send its own activation message on to its parent. After activating an update, the router runs the BGP decision process in the normal way to decide which of its active routes, received from different BRs, it should install in its FIB.

SOUP's behavior is simple so long as only one update from a BR propagates at a time: good news forward activates and bad news backward activates. At no time does the existence or absence of an alternative route received from another BR change this dissemination process. But what happens when more than one change propagates simultaneously from the same BR?

Suppose a BR has already announced route ρ_1, then receives a route change from eBGP indicating the route got worse, becoming ρ_2. It sends an update containing ρ_2, which will backward activate. Before the activation messages have returned, the route improves somewhat, so the BR sends an update containing ρ_3, which will also backward activate—even though it is better than ρ_2, it is worse than the active route ρ_1. Each router therefore maintains a list containing $\{\rho_1, \rho_2, \rho_3\}$, where ρ_1 is still active, and ρ_2 and ρ_3 are awaiting backward activation.

At the leaves of the tree, ρ_2 has now activated, and the activation for ρ_2 spreads back up the tree. Descendants on the RFT below the backward activation only have ρ_2 in their list. At some point the update for ρ_3 propagating away from the BR passes the activation message for ρ_2 returning toward the BR. On the link where these messages cross, the child router, R_c receives update ρ_3. ρ_3's update is marked as *backward activated*, but is preferable to the active route ρ_2. It is therefore feasible, is applied immediately, and is sent on marked as *forward activated.* However, as ρ_3 was received at R_c marked as *backward activated*, R_c must also indicate to its parent that it has activated the route, so it sends an activation message.

These rules ensure that an update sent as forward activated is always propagated as forward activated, but an

[2] A withdrawal is just an extreme form of a route becoming less preferred, so we will only refer to updates from now on.

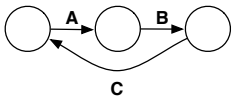


Figure 5: A simple routing loop.

update sent as backward activated may switch to forward activated if it crosses into a region where backward activation has already removed an older version of that same route. An update can only change from backward activated to forward activated once, and never the other way.

Limitations of SOUP SOUP is loop-free when disseminating changes if the IGP is loop free and internal routes do not change. It does not attempt to be loop-free while the IGP is reconverging, but as external route changes greatly outnumber internal ones in most networks, this seems a reasonable compromise.

SOUP cannot always prevent transient loops in the presence of the BGP MED attribute, though it will always converge to the same stable loop-free solution as full-mesh iBGP. With MED, although routes can be ranked as better or worse when originating from the same BR, the same is no longer true when they originate from different BRs. MED has the property that with three routes, *A*, *B*, and *C* for the same prefix, where three routers have routes *ABC*, *BC*, and *AC*, as might happen when routes *A* and *B* are propagating, then the three routers can choose routes *A*, *B*, and *C* respectively. For example, *B* beats *C* on MED, *A* beats *B* on router ID, *C* beats *A* on router ID. There is no total order among the three routes that all routers agree on, so looping can occur. iBGP-RR can exhibit persistent route oscillations in this case, and Cisco provides an *always-compare-MED* option to impose a strict total order. We conjecture that no distributed protocol can ensure loop-freedom when MED is used without such a workaround.

4.4 Proof of SOUP's Loop Freedom

To show that SOUP does not introduce forwarding loops, we focus on how the protocol handles updates for a single prefix. Our proof rests on two assumptions: first, that the IGP is not currently reconverging; and second, that BGP's always-compare-MED option is in use (such that there is a strict total order on BGP routes). Without these assumptions, SOUP cannot always prevent transient loops. We proceed in two steps: first, we introduce a sufficient condition for loop avoidance, and second, we prove that SOUP always complies with that condition.

Lemma. *When the quality of the route used for forwarding improves monotonically at all successive router hops, no forwarding loop can occur.*[3]

Proof. Consider the simple three-router topology in Fig. 5, in which letters denote the routes on which each router forwards. Define the operator "$\prec$" such that for routes x and y, $x \prec y$ means that the BGP decision process prefers y to x. Consider a path whose first edge (route) is A, along which route quality monotonically increases. We proceed by contradiction. Assume that a loop occurs along this path, and without loss of generality, assume the loop occurs between the third and first router, as shown in Fig. 5. But if this loop occurs, we have that $A \prec C$ (by the path's monotonic improvement) and also that $C \prec A$, as the loop will forward successively on routes C and A. In general, for any path on which routes improve monotonically, a forwarding loop will cause a contradiction in which the route that closes the cycle must simultaneously be less preferred and more preferred than the immediately subsequent route. □

Theorem. *SOUP does not introduce forwarding loops.*

Proof. When a packet is forwarded, the routers along the path do not have to forward towards the same BR. Some may have received new state that the others have not yet heard. However, as shown in the above lemma, so long as the routing state along the path monotonically improves, no loop can occur. For a loop to occur, monotonicity must be violated: somewhere, a router r_{n+1} must forward towards BR r_0, and the next hop, r_n, must forward using less good state than that used by r_{n+1}. Let the route being used at r_n be ρ_n* and the route being used at r_{n+1} be $\rho_{n+1}*$. (We will use $*$ to indicate that a route is installed in the FIB and used for forwarding.) To violate monotonicity, and thus for a loop to be possible, $\rho_n* \prec \rho_{n+1}*$. There are two cases to consider, depending on which BR originated ρ_n*:

Case 1: $\boxed{\rho_n* \text{ is a route that originated at } r_0.}$

For r_{n+1} to have route $\rho_{n+1}*$, it must previously have received a forward activated update for a route ρ_{n+1}^{best} from r_0 that was either better than $\rho_{n+1}*$ or is actually $\rho_{n+1}*$. This is the case because forward activated updates are the only way routes can improve. As r_{n+1} is the child of r_n with respect to BR r_0, for r_{n+1} to have received ρ_{n+1}^{best}, it must have received this route from r_n. r_n must therefore have also held ρ_{n+1}^{best} at some point.

If $\rho_n* \prec \rho_{n+1}*$, then r_n must have received an update that replaced ρ_{n+1}^{best} with a worse route. Such an update must have backward activated at r_n, as it would not have been feasible compared to ρ_{n+1}^{best}. However, for ρ_n* to have been backward activated, as r_{n+1} is the child of r_n, the activation must have passed through r_{n+1}, and it would have replaced $\rho_{n+1}*$. Thus r_{n+1} cannot have $\rho_{n+1}* \succ \rho_n*$ if ρ_n* originated at r_0.

To summarize: if r_{n+1} has route ρ_{n+1}, then its parent r_n must still have the same route or a better one.

Case 2: $\boxed{\rho_n* \text{ originated at BR } r_{BR}, \text{ where } r_{BR} \neq r_0.}$

However, by case 1, if r_{n+1} has route $\rho_{n+1}*$ received from r_0, then r_n must still have a route $\rho_n^{r_0}$ received

[3] Jaffe and Moss offer a similar proof [16].

from r_0, such that $\rho_{n+1}* \prec \rho_n^{r_0}$. If router r_n has route $\rho_n^{r_0} \succ \rho_n*$, then it will not choose ρ_n*—the BGP decision process can only choose $\rho_n* \succ \rho_n^{r_0}$. Hence it cannot be the case that $\rho_n* \prec \rho_{n+1}*$.

As neither case 1 nor case 2 can occur, it is impossible for a loop to occur. □

5 Link-Ordered Update Protocol

SOUP avoids loops by always maintaining the invariant that no router ever has better active state for a prefix from a specific BR than its parent router. The down side of maintaining this invariant is that bad news must normally propagate all the way across the network and the backward activations return before worsening or withdrawn state can be removed. If, after receiving bad news from eBGP, a BR still has an external route, even though it is no longer the best route from the domain, then there is no significant problem: the destination is still reachable, and all that happens is a suboptimal path is used for a short while. However, when the BR receives a withdrawal via eBGP and has no other eBGP route, then it will have to drop any packets for this destination. SOUP forces the BR to hold the old route longer than we would wish, prolonging the black hole longer than alternative protocols that are not loop-free.

Is it possible to get the best of both worlds: maintain loop-free routing, but also reconverge quickly? Backward activation of withdrawals is essential for loop-free routing, but inevitably delays switching to an alternative route. Thus SOUP cannot converge as fast as protocols that do not perform backward activation. However, it is often safe to terminate a backward activation without having it traverse the whole network and back.

Consider Fig. 3, where the route advertised by B is being withdrawn and an alternative route from A that tie-broke on IGP distance exists. To avoid loops, SOUP's withdrawal wavefront spreads the whole way across the network before activating on the reverse path. It is safe instead to activate the withdrawal as soon as the wavefront reaches ⑤. The first hop to the left of ⑤ could trigger the backward activation of the withdrawal by sending a reply. This short-cutting of backward activation is the essence of the Link-Ordered Update Protocol (LOUP).

5.1 Local Activation of Withdrawals

Just as with SOUP, a LOUP router passes a worsening update on to its children without activating it, and waits for backward activations from them before activating the route change. Doing so maintains the invariant that a router never has a better activated version of the route than its parent does. This invariant ensures that successive announcements of a route from the same BR cannot cause a loop.

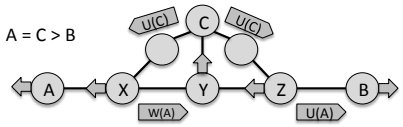


Figure 6: Local activation can lead to transient loops.

Fig. 3 might lead us to consider that the routers just to the left of ⑤ could locally activate the withdrawal. They are not using the route being withdrawn, so they could reply immediately to their parent, activating the withdrawal back toward B. In the steady state, doing so is safe and does not cause loops. However, it violates the invariant that no router ever has a better activated version of a route than its parent. Although it is quite difficult to find cases where a loop results, they do exist.

Consider Fig. 6. The three BRs, A, B, and C have all announced routes, such that $A = C \succ B$. The updates from A and C have not yet reached B, so B has not yet withdrawn its route. C's update has reached Y, but not reached X and Z. A has withdrawn its route—the withdrawal will backward activate. Thus A, X, and Z are currently using route A (shown by the block arrows), Y is using C, and B is using its own route. So far, the network is loop-free.

The withdrawal of A then reaches Y, which is not using route A. Y therefore activates the withdrawal, passes it to its children, and sends an activation message back to X. Before any of these messages are received, Y then receives a withdrawal of C. Because neither X nor Z reaches C via Y, Y is a leaf on C's RFT. Y immediately activates the withdrawal, and switches to its only alternative, which is B. Traffic now loops between Y and Z.

Thus we can see that local activation is not safe in the presence of transient announcements such as the one from C. The problem is that the existence of the route from C causes routers Y and Z to violate the invariant—Z still has the route to A, but Y does not. To maintain the invariant and be loop-free, the routing state at a route for one BR must not depend on the existence or absence of routing state for another BR.

5.2 Targeted Tell-me-when

To avoid the problem associated with local activation, we build upon the sound basis of SOUP. LOUP activates an update exactly as SOUP does—it triggers backward activation under the same conditions as SOUP. However, if a router has an alternative route via another neighbor, it is safe to switch to that route if the router knows that neighbor is no longer using the route that is waiting for a backward activation.

Upon receiving a withdrawal (or a worsening update), a router marks the route as backward activated and passes it on to its children. It then also runs the decision process to determine which route it would use if the change

were activated. If there is an alternative route via another neighbor, the router sends a *tell-me-when* request to that neighbor requesting that it reply when it is no longer using the route being withdrawn.

If the response to the *tell-me-when* arrives before the activation, the router knows that it is now safe to use the alternative because its new next hop (and transitively, all routers between it and the new exit point) is already using the alternative. The router does not yet activate the withdrawal, but it can reply to any *tell-me-when* from its other neighbors.

If the alternative route is withdrawn, the router still has the original route (waiting for backward activation of the withdrawal) in its table, and can safely switch back to it if necessary as the invariant still holds, so long as it has not replied to any other router's *tell-me-when* for this route. If it has replied to a *tell-me-when*, the route is marked as *dead*. It is unsafe to switch back to a dead route, and packets for this prefix will be black-holed until the backward activation arrives.

Only when the backward activation arrives from all a router's children is a withdrawal finally activated.

5.3 Safety

With respect to a BR, LOUP maintains the same invariant as SOUP, so no two routers forwarding on state from the same BR can cause a loop. In [13], for an older version of LOUP that used targeted tell-me-when messages but did *not* use backward activation, we exhaustively considered all the possible ways a single update or withdrawal could interact with existing forwarding state at routers. So long as the IGP itself is loop-free, there was only one way a loop could occur: a race condition we called a Withdrawal/Announcement race, where a withdrawal of one route caused a previously suppressed route to be reannounced. The triggering withdrawal and the triggered announcement could race, leading to loops. The current LOUP protocol's backward activation mechanism prevents this race condition. We thus assert that no single update can cause LOUP to create a forwarding loop. We make no assertion that LOUP is loop-free when a BR sends multiple updates for the same prefix in extremely rapid succession, but we have not seen such loops in simulation. BGP's MRAI timer would normally prevent this.

5.4 Freedom from Configuration Errors

Full-mesh iBGP requires all peerings be configured. The configuration is simple, but all routers must be reconfigured whenever any are added or removed. Route reflectors and confederations add configured structure to an AS, and require expert knowledge to follow heuristics to avoid sub-optimal routing or persistent oscillations. A BGP-free core improves iBGP's scaling somewhat, at the expense of requiring additional non-trivial mechanisms just to route traffic across the network core. All this configuration significantly increases the likelihood of disruptions caused by configuration errors.

Hop-by-hop dissemination mechanisms such as BST and LOUP are configuration-free. All one must do is enable the protocol. Some might equate configuration with control. We will show that LOUP's freedom from configuration does not give rise to routing protocol traffic hotspots.

6 Building and Using the RFT

We now describe the details of ordered update dissemination along an RFT in the SOUP and LOUP protocols.[4] There are two main aspects: how to build the RFT, and how to disseminate updates along the RFT reliably despite topology changes.

6.1 RFT Construction

Each LOUP router derives a unique ID (similar to BGP-ID) that it uses to identify routes it originates into the AS. LOUP routers periodically send single-hop link-local-multicast Hello messages to allow auto-discovery of peers. A Hello contains the sender's ID and AS number. Upon exchanging Hellos containing the same AS number, a pair of LOUP routers establish a TCP connection for a peering. All LOUP protocol messages apart from Hellos traverse these TCP connections, and are sent with an IP TTL of 1.

A LOUP router must know the IDs of all LOUP routers in its AS to build and maintain the RFTs. This list is built by a gossip-like protocol that operates over LOUP's TCP-based peerings. Essentially, a LOUP router announces the full set of LOUP router IDs it knows to its neighbors each time that set grows (and to bootstrap, it announces its own ID when it first peers with a neighbor). These gossip messages need not be sent periodically, as they are disseminated reliably with TCP. LOUP routers time out IDs from this list upon seeing them become unreachable via the IGP.

The RFT rooted at a router X is the concatenation of the forwarding paths from all routers to X—the *inverse* of the relevant adjacencies in routers' routing tables. Whenever the RFT changes, each LOUP router sends each of its neighbors a Child message. LOUP router Y will send its neighbor X a Child message stating, "you are my parent in the RFT for this set of IDs." This set of IDs is simply the set of all IGP-learned destination IDs in Y's routing table with a next hop of X. Upon receiving a Child message on interface i, LOUP router X subsequently knows that it should forward any message that *originated* at any ID mentioned in that Child message down the appropriate RFT on interface i.

[4] Both protocols use the exact same RFT techniques; we write "LOUP" hereafter for brevity.

6.2 Reliable RFT Dissemination

An *origin* LOUP router that wishes to send an update (*e.g.,* a BR injecting an update received over eBGP) sends that update to all routers in its AS over the RFT rooted at itself. LOUP routers forward such updates over their one-hop TCP peerings with their immediate neighbors on the appropriate RFT. During a period when no topology changes occur in an RFT, TCP's reliable in-order delivery guarantees that all updates disseminated down the RFT will reach all routers in the AS.

When the topology (and thus the RFT) changes, however, message losses may occur: if the distance between two routers that were previously immediate neighbors changes and exceeds a single hop, the IP TTL of 1 on the TCP packets LOUP sends over its peerings will cause them to be dropped before they are delivered. For RFT-based update dissemination to be reliable under topology changes, then, some further mechanism is needed.

To make update dissemination over the RFT robust against topology changes, the LOUP protocol structures updates as a *log*. Each router maintains a log for each origin. A log consists of an active operation and one or more inactive operations, each with a sequence number, ordered by these sequence numbers; this sequence number space is per-origin. An operation may either be a route update or a route withdrawal. Inactive operations are backward propagated updates that have not yet been activated. When a LOUP router receives an operation for dissemination over the RFT on a TCP peering with a neighbor, it only accepts the operation and appends it to the appropriate origin's log if that operation's sequence number is one greater than that of the greatest sequence number of any operation already in that origin's log. That is, a router only accepts operations from an origin for RFT dissemination in contiguous increasing sequence number order.

Should a LOUP router ever receive an operation for RFT dissemination with a sequence number other than the next contiguous sequence number, or should a temporary partition occur between erstwhile single-hop-neighbor routers, LOUP may need to recover missing operations for the origin in question. A LOUP router does so by communicating the next sequence number it expects for each origin's log to its current RFT parent. LOUP includes this information in Child messages, which routers send their parents for RFT construction and maintenance, as described above. Should an RFT parent find that it holds operations in a log that have not yet been seen by its RFT child, it forwards the operations in question to that child.

LOUP requires that the topology within an AS remains stable long enough for LOUP to establish parent-child adjacencies with its Child messages. So long as this condition holds, LOUP's single-hop TCP connections coupled with its log mechanism guarantee reliable dissemination of operations down the RFT. Topology changes may temporarily disrupt the RFT, but all data will eventually reach the entire AS.

When a BR wishes to distribute a route update or withdrawal, it acts as an origin: it adds this operation to its log with the next unused sequence number, and sends it down the RFT. As routers receive the operation, they apply it to their logs. When all operations are eventually activated the end effect is the same as that of full-mesh iBGP because the origin BR disseminates its update or withdrawal to every router in the AS, just as full-mesh iBGP does.

7 EVALUATION

We evaluated SOUP and LOUP to examine their correctness, scalability, and convergence speed. To be correct they must:

- always converge to the same solution as full-mesh iBGP—doing so guarantees no persistent oscillations if eBGP policy obeys AR; and
- not create transient loops, so long as the underlying IGP's routes are loop-free.

We assess scalability by asking:

- How is the CPU load distributed between routers?
- How are FIB changes distributed? Is the FIB updated more frequently than with iBGP?
- How much churn is propagated to neighboring ASes?
- What is the actual cost of processing updates? Can bursts of updates be handled quickly enough?
- Can the implementation hold the global routing table in a reasonable memory footprint? Neither SOUP nor LOUP hides information, so how well does it compare to BGP with RRs?

Finally, we consider the delay and stability behavior of these protocols during convergence, and compare them with the alternate loop-free strategy of injecting external routes into DUAL:

- How do the convergence times of SOUP and LOUP compare? How long does DUAL take to converge when conveying external routes?
- Does injecting external routes into DUAL render the network unstable? What is the cost when an internal link comes up or down?

7.1 Methodology

We implemented LOUP, SOUP, iBGP with RRs and a generic flooding protocol that we will call BST* in a purpose-built event-driven network simulator.[5] We also implemented a version of DUAL that injects external

[5]We wanted to implement BST, but there is no clear spec, so it probably differs from BST in some respects.

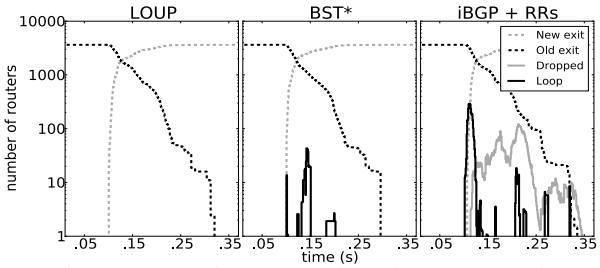


Figure 7: Transients on update (less connected)

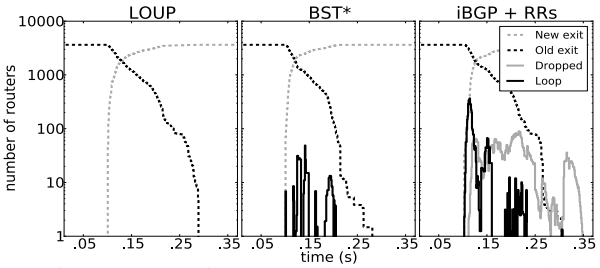


Figure 9: Transients on update (more connected)

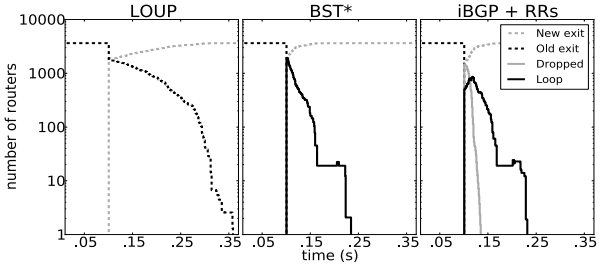


Figure 8: Transients on withdrawal (less connected)

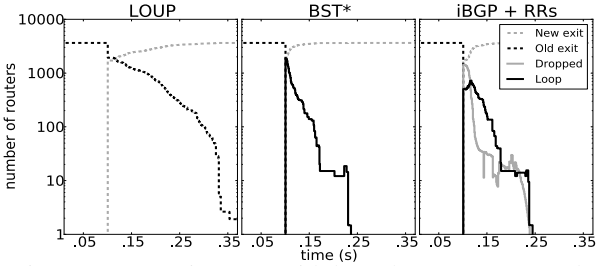


Figure 10: Transients on withdrawal (more connected)

routes, so it can be used as a loop-free iBGP replacement. In addition we implemented LOUP for Quagga [14], and compare it to Quagga's iBGP implementation.

We have explored the behavior of SOUP and LOUP on a wide range of synthetic topologies, including grids, cliques, and trees. These scenarios included varying degrees of link and router failure and the presence of MED attributes. In all cases the two protocols required no explicit configuration and converged to the same solution as full-mesh iBGP.

To illustrate the behavior of the protocols in a realistic scenario, we simulate a network based on that of Hurricane Electric (HE), an ISP with an international network. We use publicly available data: HE's backbone topology (including core router locations) and iBGP data that reveal all next hops in the backbone [1]. These next hops are the addresses of either customer routers or customer-facing routers. We assume there is an attachment point in the geographically closest POP to each distinct next hop, create a router for each attachment point, and assign it to the closest backbone router. For iBGP-RR, we place RRs on the core routers and connect them in a full mesh. Recent studies suggest this model is not unrealistic [4].

We explore two different levels of redundancy. In the baseline redundancy case all clients in a POP connect to two aggregation routers, which in turn connect to the core router. In the more redundant case each aggregation router is additionally connected to the nearest POP. Unless explicitly specified all simulation results are from the more connected case.

We model speed-of-light propagation delay and add a uniform random processing delay in [0, 10] ms. We do not, however, model queues of updates that might form in practice, so our simulations should produce shorter-lived transients than might be seen in real backbones.

In the case of DUAL, we inject eBGP routes into DUAL by mapping the BGP path attribute list to a DUAL metric, and then allow DUAL to distribute these routes internally as it normally does.

7.2 Correctness

To examine transient loops we compare the behavior of LOUP, BST*, and iBGP in two scenarios involving a single prefix: the announcement of a new "best" route for a prefix, and the withdrawal of one route when two routes tie-break on IGP distance. We compare both the less redundant and the more redundant topologies to observe the effect of increased connectivity. Figs. 7 and 9 show the protocols' behavior when a single BR propagates an update, and all routers prefer that update to a route they are already using for the same prefix. As a result, this update triggers a withdrawal for the old route. And Figs. 8 and 10 show the protocols' behavior in the tie-break withdrawal case.

We are interested in how the prefix's path from each router evolves over time. Define the correct BR before the change occurs as the old exit and the correct BR after the change occurs and routing converges as the new exit. In these four figures, we introduce the initial change at time $t = 0.1$ seconds and every 100 μs we check every router's path to the destination. Either a packet correctly reaches the new BR, still reaches the old BR, is dropped, or encounters a loop. The y-axis shows the number of routers whose path has each outcome. We plot the mean of 100 such experiments, each with randomly chosen BRs as the old and new exits.

Figs. 7 and 9 confirm that LOUP incurs no transient loops or black holes and its convergence time is similar to

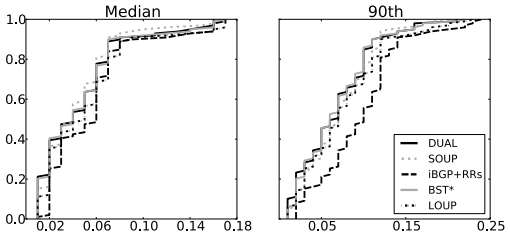


Figure 11: Convergence delay on update (sec)

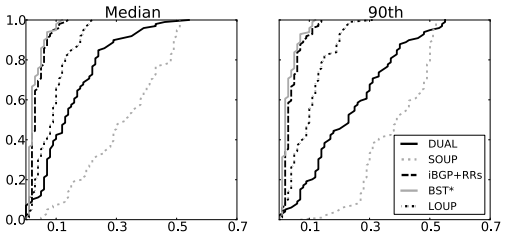


Figure 12: Convergence delay on withdrawal (sec)

that of the other protocols. BST* and iBGP-RR perform as expected; BST* does not cause black holes, but IBGP-RR causes both loops and black holes. On the less connected topology, there is limited opportunity for races to propagate far, so BST incurs relatively few loops. When it does loop, many paths are affected - the BST results have high variance. The more redundant the network, the more opportunity there is for BST to cause loops, as is evident from Fig. 9.

Figs. 8 and 10 demonstrate the importance of enforcing ordering on withdrawals. LOUP does not cause loops, but it takes longer to converge because the withdrawal first must propagate to the "tie" point and then be activated along the reverse path. All other protocols loop transiently because the BR immediately applies the withdrawal resulting in a loop like that in Fig. 3.

In the interest of brevity we omitted graphs for DUAL and SOUP, but neither incurred transient loops. We examine their convergence latency separately below.

7.3 Convergence Delay

How quickly do the various protocols propagate changes to all routers? We repeated the single-update and single-withdrawal experiments from Fig. 10 for 100 passes per protocol. We collected the median and 90th percentile delays of all passes and present their CDFs. We also present results for DUAL and SOUP. The results are in Figs. 11 and 12. When disseminating good news, all the protocols incur similar delay.

Fig. 12 shows the price paid to avoid looping with bad news. BST and iBGP converge fastest, but cause transient loops. SOUP performs worst, as it cannot short-cut the activation of withdrawals, propagating them all the way to the end of the AS, and only then activating. In fact, if one or two routers were slower, this effect would be exacerbated. Both LOUP and DUAL overcome this

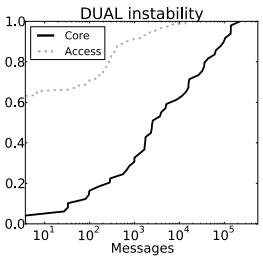


Figure 13: DUAL instability

problem, but DUAL's flooding nature means that it is a little slower to converge. When frozen, each DUAL router must collect replies from all its neighbors before terminating the diffusing computation, while a LOUP router only needs to wait for its children on the RFT. This is the reason for the gap between DUAL and LOUP.

7.4 Stability

The most significant problem with distributing external routes into DUAL is the increased cost of IGP changes. After LOUP, SOUP or BST has propagated external routes through the network, any route change in the IGP will not cause additional churn—the IGP will reconverge, but there will be no need to re-exchange external routing information. If a link fails with DUAL, a new path is calculated for each destination, including the external ones, as each may be injected from a different subset of BRs. To do this requires a great deal of communication. In some cases a single link's failure may cause all external routes to be frozen and DUAL will have to re-converge for each of them.

We injected 5000 routes from the HE data set into DUAL—these relate to 300 different prefixes. We then failed one link and observed the traffic generated that relates to these external prefixes. We repeated this process for every core link and 300 randomly chosen access links. Fig. 13 shows a CDF of the results. For 50% of the access links, a failure results in more than 2000 messages being exchanged and 10% of link failures generate more than 10000 messages. We would expect message complexity to scale linearly; for example, with a full routing table of 300K prefixes, we would expect 50% of core link failures to result in more than 2 million messages being sent.

Access link failures generate fewer messages—60% generate none in this experiment because these BRs injected no external routes. With a full routing table, all these would have injected some routes. Of the ones that were on the shortest-path tree for some external prefix, 50% of link failures generated more than 300 messages. With a full routing table, we would expect this number to grow to around 300K messages.

We have omitted results for LOUP, SOUP and BST because they generate no routing messages for exter-

nal prefixes when the IGP changes, though LOUP and SOUP generate up to one message per neighbor to maintain the RFTs. iBGP does generate some churn as RRs change preferred routes and inform their clients of the changes, but this churn is usually insignificant compared to DUAL's, and it does not manifest in this scenario.

7.5 Scalability

To what extent do the different protocols concentrate processing load in a few routers? We take a set of 10000 routes from HE's iBGP data set, taking care to preserve all alternatives for each prefix we select, and inject them rapidly into the simulated 3000 router HE network. We rank the routers in terms of messages sent or received, and show the 750 busiest in Figs. 14, 15, and 16. Message counts are a measure of communication cost, and update counts are a measure of the cost of running the BGP decision process.

BST's flooding means it incurs greater communication cost and processes many updates. LOUP and SOUP are a little more expensive than iBGP, as route reflectors hide some updates from their clients. As the HE route set does not include many external withdrawals, tell-me-when kicks in rarely, so LOUP and SOUP perform almost identically. DUAL's overall costs are similar, but it performs more processing in well-connected core routers as it explores alternatives.

Control-plane overhead is only one aspect to scalability: on some routing hardware, FIB changes are the routing bottleneck [6]. Generally FIB adds or deletes where the trie may need to be rebalanced are more costly than in-place updates to existing entries.

Figs. 17 and 18 show FIB operations during the experiment above. All protocols except DUAL perform a similar number of operations. Although iBGP processes fewer messages, those messages are more likely to cause expensive FIB adds or deletes. Route reflectors hide information. Doing so can lead to path exploration during which the FIB is modified multiple times, but it may also shield RRs' clients from a number of FIB updates. SOUP, LOUP, and BST exhibit virtually identical behavior because they exchange the same information.

The DUAL results show the number of times DUAL changes successor. It can do so often, as its loop-avoidance mechanism needs to freeze and then unfreeze portions of the network when updates for the same prefix propagate at the same time.

Fig. 19 shows FIB operations at the BRs only. When a BR changes route, it usually notifies its external peers, so this is a measure of churn passed on to eBGP. DUAL is significantly worse than the other protocols here, as it explores more alternatives before converging.

To evaluate CPU usage we run our Quagga-based LOUP implementation using a simple topology consisting of three single-core 2Ghz AMD machines (*A*, *B* and *C*) connected with gigabit links. In BGP's case we open an eBGP session from *A* to *B* and an iBGP session from *B* to *C*. In LOUP's case we perform no configuration. We inject one view of the global routing table ($\tilde{4}$00,000 routes) at *A*, which forwards to *B*, which forwards to *C*. We look at the load on *B* as it must both receive and send updates, and does the most work.

Task	LOUP	BGP
Updating the RIB	1981	5042
Updating the FIB	6544	16874
Serialization	3222	7477
Low-level IO	7223	6447
Other	2824	5369
Total (million cycles)	21797	41212
Total (seconds)	10.8	20.6

Both protocols spend most of the time updating the FIB and doing low-level IO. Running the decision process and updating the RIB data structures is almost negligible. LOUP is much faster than BGP, but it seems that Quagga's BGP spends unnecessary time updating the FIB, so this effect is not fundamental.

LOUP's memory usage, below, depends directly on the number of routes for a prefix that tie-break on IGP distance, as other alternatives will be withdrawn.

BGP (1)	LOUP (1)	LOUP (2)	LOUP (3)
73.2 MB	46.7 MB	68.2 MB	89.8 MB

Memory usage is shown when we injected the same route feed from 1, 2 and 3 different BRs in our experimental network. We only present results for BGP with one view, because the RRs hide all but the winning routes from their clients. Because BGP has to maintain multiple RIBs for each session, its memory footprint is greater than LOUP's. Based on HE's data, in a large ISP there will be on average 5-6 alternatives for a prefix. LOUP's memory usage grows linearly, so we expect the protocol to run easily in a network like HE's on any modern router with 200 MB or more of RAM.

8 Related Work

There has been significant work on carefully disseminating routing updates so as to improve the stability of routing and ameliorate pathologies such as loops and black holes. We have discussed DUAL's approach to loop-free IGP routing [8], BST's reliable flooding approach to intra-AS route dissemination [15], and RCP's centralized approach to intra-AS route dissemination [3] at length in Sections 2 and 3. To recap: LOUP tackles loop-free intra-AS dissemination of externally learned routes, a different problem than loop-free IGP routing, as taken on by DUAL and oFIB [22]; the non-convexity of BST's flooding causes transient loops that LOUP avoids; and RCP centralizes the BGP decision process for an AS, but does

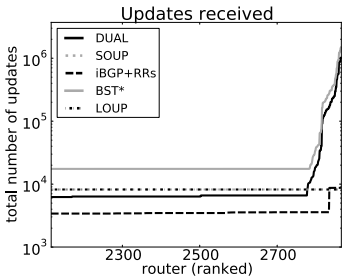


Figure 14: Updates received

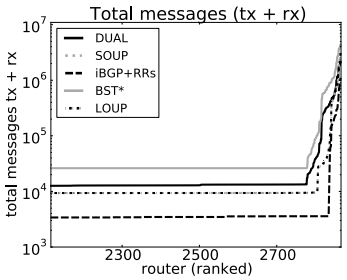


Figure 15: Total messages

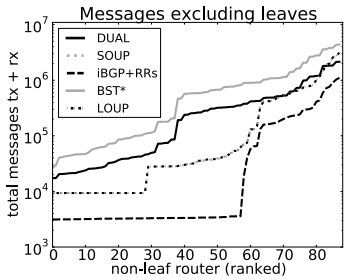


Figure 16: Messages (no leaves)

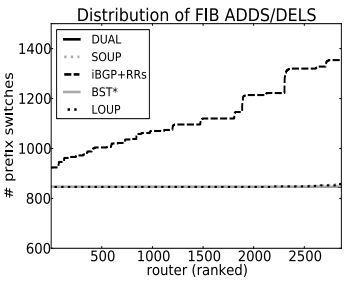


Figure 17: FIB adds and deletes

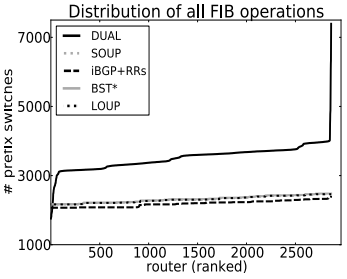


Figure 18: FIB operations

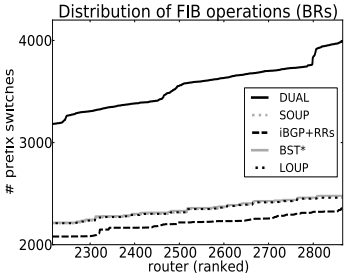


Figure 19: FIB operations (BRs)

not propagate the results synchronously to all routers, and so does not achieve the freedom from transient loops and black holes that LOUP does. We note that loop-free IGPs like DUAL (and its implementation in EIGRP) complement LOUP nicely: running LOUP *atop* DUAL would prevent both IGP loops and transient loops present in today's route dissemination by iBGP with RRs.

Consensus Routing [17] adds Paxos-based agreement to eBGP to avoid using a route derived from an update until that update has propagated to all ASes. LOUP's ordered, reliable dissemination of updates along an RFT is lighter-weight than Paxos-based agreement, yet still avoids introducing loops within an AS during dissemination. Bayou's logs of sequence-number-ordered updates [24] and ordered update dissemination [21] inspired the analogous techniques in LOUP; we show how to apply these structures to achieve robust route dissemination, rather than weakly consistent storage.

In prior work [13], we first proposed ordered, RFT-based dissemination as a means to avoid transient loops. In this paper, we have additionally described SOUP and LOUP, full routing protocols built around these principles, proven that SOUP never causes forwarding loops, and evaluated the scalability of a full implementation of LOUP atop the Quagga open-source routing platform.

9 Conclusion

The prevalence of real-time traffic on today's Internet demands greater end-to-end path reliability than ever before. The vagaries of iBGP with route reflectors—transient routing loops, route instability, and a brittle, error-prone reliance on configuration—have sent network operators running into the arms of MPLS, in an attempt to banish iBGP and its ills from the core of their networks. In exploring the fundamental dynamics of route dissemination, we have articulated why iBGP with route reflectors and BST introduce such pathologies. Based on these fundamentals, we have described a simple technique—ordered dissemination of updates along a reverse forwarding tree—that avoids them. And we have illustrated how to apply this technique in practice. SOUP is provably loop-free, but incurs latency associated with network-wide backward activation of less preferable routes. LOUP converges faster than SOUP by short-cutting backward activation in common cases. During convergence after a single update from a single BR, LOUP prevents forwarding loops. But as LOUP may incur loops under heavy update churn for the same prefix from multiple BRs, it trades absolute loop-freedom for faster convergence. Our evaluation in simulation has revealed LOUP to be a practical, scalable routing protocol, which we have also seen through to a prototype implementation for Quagga. While earlier work has drawn upon consistency techniques from the distributed systems community to improve the robustness of routing, SOUP and LOUP achieve strong robustness with lighter-weight mechanisms. As such, we believe they offer compelling alternatives to a BGP-free core.

Acknowledgements

We thank Arvind Krishnamurthy, our shepherd, and the anonymous referees for their insightful comments. This research was supported by EU FP7 grants 287581 (OpenLab) and 257422 (Change), and by a gift from Cisco.

REFERENCES

[1] Hurricane electric's looking glass server. `route-server.he.net`, July 2012.

[2] L. Andersson, I. Minei, and B. Thomas. LDP specification. *RFC 5036*, Oct. 2007.

[3] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and K. van der Merwe. Design and implementation of a routing control platform. In *Proc. ACM/USENIX NSDI*, 2005.

[4] J. Choi, J. H. Park, P. chun Cheng, D. Kim, and L. Zhang. Understanding BGP next-hop diversity. *IEEE INFOCOM*, Apr. 2011.

[5] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[6] K. Fall, G. Iannaccone, S. Ratnasamy, and P. Godfrey. Routing tables: Is smaller really much better? *Proc. ACM HotNets*, 2009.

[7] L. Gao, T. Griffin, and J. Rexford. Inherently safe backup routing with BGP. *IEEE INFOCOM*, 2001.

[8] J. Garcia-Luna-Aceves. Loop-free routing using diffusing computations. *IEEE/ACM Transactions on Networking*, 1(1), Feb. 1993.

[9] T. Griffin and G. Wilfong. An analysis of BGP convergence properties. *SIGCOMM*, 1999.

[10] T. Griffin and G. Wilfong. Analysis of the med oscillation problem in BGP. *ICNP*, 2002.

[11] T. Griffin and G. Wilfong. On the correctness of IBGP configuration. In *Proc. SIGCOMM 2002*, Aug. 2002.

[12] T. Griffin and G. T. Wilfong. Analysis of the MED oscillation problem in BGP. In *Proc. ICNP '02*, pages 90–99, Washington, DC, USA, 2002.

[13] N. Gvozdiev, B. Karp, and M. Handley. LOUP: who's afraid of the big bad loop? *Proc. ACM HotNets*, Oct. 2012.

[14] K. Ishiguro et al. Quagga, a routing software package for TCP/IP networks. `http://www.quagga.net`, Mar. 2011.

[15] V. Jacobson, C. Alaettinoglu, and K. Poduri. BST - BGP scalable transport. *Presentation at NANOG 27*, Feb. 2003.

[16] J. Jaffe and F. Moss. A responsive distributed routing algorithm for computer networks. *IEEE Transactions on Communications*, 30(7):1758–1762, 1982.

[17] J. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataraman. Consensus routing: The internet as a distributed system. In *Proc. NSDI 2008*, Apr. 2008.

[18] N. Kushman, S. Kandula, and D. Katabi. Can you hear me now?! It must be BGP. *ACM CCR*, Apr. 2007.

[19] D. McPherson, V. Gill, and D. Walton. Border gateway protocol (BGP) persistent route oscillation condition. *RFC 3345*, Aug. 2002.

[20] J. H. Park, R. Oliveira, S. Amante, D. McPherson, and L. Zhang. BGP route reflection revisited. *IEEE Communications Magazine*, July 2012.

[21] K. Petersen, M. J. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In *Proc. SOSP 1997*, Oct. 1997.

[22] M. Shand, S. Bryant, S. Previdi, C. Filsfils, P. Francois, and O. Bonaventure. Loop-free convergence using ofib. *IETF Internet Draft draft-ietf-rtgwg-ordered-fib-06*, June 2012.

[23] V. Srinivasan and V. G. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems*, 1999.

[24] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc SOSP 1995*, Dec. 1995.

Improving availability in distributed systems with failure informers

Joshua B. Leners* Trinabh Gupta* Marcos K. Aguilera† Michael Walfish*

*The University of Texas at Austin †Microsoft Research Silicon Valley

Abstract. This paper addresses a core question in distributed systems: how should applications be notified of failures? When a distributed system acts on failure reports, the system's correctness and availability depend on the granularity and semantics of those reports. The system's availability also depends on coverage (failures are reported), accuracy (reports are justified), and timeliness (reports come quickly). This paper describes *Pigeon*, a failure reporting service designed to enable high availability in the applications that use it. Pigeon exposes a new abstraction, called a *failure informer*, which allows applications to take informed, application-specific recovery actions, and which encapsulates uncertainty, allowing applications to proceed safely in the presence of doubt. Pigeon also significantly improves over the previous state of the art in the three-way trade-off among coverage, accuracy, and timeliness.

1 Introduction

Availability is now a paramount concern of distributed applications in data centers and enterprises (distributed storage systems, key-value stores, replication systems, etc.); for such applications, even seconds of downtime can affect millions of users. A critical factor in availability is failure handling. Specifically, for optimal availability, distributed applications need to learn of failures quickly, so that they can recover, and they need information *about* the failure, so that they can take the best recovery action.[1]

This paper proposes *Pigeon*, a service for reporting host and network failures to highly available distributed applications. Pigeon provides a new abstraction, called a *failure informer*. This abstraction hides the messy details of failures; it reports a small number of conditions that each represent a class of problems that affect the application similarly. The conditions are differentiated by the failure certainty, or lack thereof, which gives enough information for applications to improve their recovery, in application-specific ways.

For example, if a lease server [13, 30] is informed of the certain crash of a process holding a lease, the server can bypass the lease delay and reissue the lease immediately; without this information, the lease server would have to wait until the lease times out. As another example, consider a primary-backup system [4]. If Pigeon reports to the backup that the primary has certainly stopped, the backup takes over immediately; if Pigeon reports that the primary is (possibly intermittently) unreachable, the backup must decide whether to fail over the primary, based on the expected problem duration (which Pigeon reports) and the cost of failover; and if Pigeon reports that the primary is expected to crash soon, the backup can provision a new replica without failing over the primary yet.

In the above example, notice that the different reports from Pigeon are qualitatively different and allow qualitatively different failure responses. Consider, by contrast, existing mechanisms for reporting failures, such as ICMP, TCP connection reset, and failure detectors [17] built on tuned timeouts [11, 18, 34, 62] or on layer-specific monitors [46]. These mechanisms not only cannot distinguish between various failure conditions but also have other shortcomings (as argued in Section 2.1). These shortcomings are rooted in the network's design:

> [At] the top of transport, there is only one failure, and it is total partition. The architecture was to mask completely any transient failure. . . . the Internet makes very weak assumptions about the ability of a network to report that it has failed. [The] Internet is thus forced to detect network failures using Internet level mechanisms, with the potential for a *slower and less specific error detection* [emphasis added] [21].

The rationale was simplicity. Since the network was to be designed for survivability above almost everything else [21, §3–§4], and hence would recover from failures, the benefit of exposing failures to applications was not worth the cost of a mechanism. Yet, availability of distributed applications—a more pressing concern now than it was then—calls for additional design: we want *faster and more specific error detection*!

What should such a failure reporting service look like? Answering this question requires addressing several challenges. First, there are many failure indicators (e.g., monitors reporting crashed processes, status of network links, hardware error status), each with its own idiosyncrasies, but what details should be exposed to applications? Second, these indicators may report uncertain information, leading to wrong conclusions. Addressing these two challenges requires finding the right abstraction for failure reporting—one that is simple but conveys

[1] By *failure*, we mean a problem that is visible end-to-end, not masked; by *recovery*, we mean actions in response to such failures (failover, etc.). Techniques such as microreboot and component restart [14, 15] are failure *prevention*, which is orthogonal to (in the sense that it does not obviate) our concern of failure *reporting*.

the information that lets applications recover effectively. The third challenge is in implementing the abstraction: to improve application availability, the implementation must provide full *coverage* (failures are reported), but also provide *accuracy* (reports are justified), and *timeliness* (failures are reported quickly). Meanwhile, these considerations are in a three-way trade-off.

Our response, Pigeon, classifies failures into four types: whether the problem certainly occurred versus whether it is expected and imminent, and whether the target is certainly and permanently stopped versus not. Observe that a report of certain occurrence and certain permanence abstracts "process crash" (among other things), and a report of certain occurrence and uncertain permanence abstracts "pending timeout expired" or "network partition" (among other things). Furthermore, observe that applications can benefit from even the uncertain reports: they can consider the cost-benefit trade-offs of waiting versus recovery (for problems of uncertain permanence) and of waiting versus precautionary actions (for problems of uncertain occurrence). Pigeon includes other information too, such as expected problem duration, and the resulting abstraction is what we refer to as a failure informer. To summarize the abstraction, it knows what it knows, it knows what it doesn't know, and applications benefit from hearing the difference.

Pigeon manages the conflict among coverage, accuracy, and timeliness by relying on an end-to-end timeout as a backstop (achieving full coverage) and then using low-level information from throughout the system to significantly improve the accuracy-timeliness tradeoff. The use of low-level information is inspired by Falcon [46]. However, Falcon has limited coverage (network failures cause it to hang), a coarse interface (it reports crashes only), and adverse collateral effects (it kills components, sometimes gratuitously). We elaborate on these points in Section 2.1 and compare the two systems in Section 7.

Our implementation of Pigeon has several limitations and operating assumptions. First, Pigeon assumes a single administrative domain (but there are many such networks, including enterprise networks and data centers). Second, Pigeon requires the ability to install code in the application and network routers (but doing so is viable in single administrative domains). Third, for Pigeon to be most effective, the administrator or operator must perform environment-specific tuning (but this needs to be done only once).

Before continuing, we emphasize that the challenges of Pigeon are mostly in architecture and design, as opposed to low-level mechanism; the mechanisms in Pigeon are largely borrowed from previous work [36, 37, 46, 59, 60]. The contributions of this work are:

- The thesis that network and host failures should be exposed to applications (§2). Though simple, this thesis has apparently not been advanced in previous work (§7).
- The failure informer abstraction for exposing failures (§3.1–§3.2) and a service, Pigeon, that implements it (§3.4–§3.5). As is often the case with concise but powerful abstractions, this one may appear "easy", yet identifying it was not, judging by our own repeated attempts.
- The uses of Pigeon (§3.3, §5.2). Our confidence in the abstraction is bolstered by concrete use cases.
- The evaluation (§5) of our prototype (§4). For a minor price in resources, Pigeon quickly (sub-second time) and accurately reports common failure types. Pigeon quantitatively and qualitatively outperforms other mechanisms (including Falcon), and we demonstrate that it allows real applications to make better, faster, application-specific recovery decisions.

2 Motivation, challenges, and principles

We now explain the status quo's shortcomings (§2.1) and the principles that Pigeon is based on (§2.2).

2.1 Failure reporting today

Existing mechanisms for reporting failures are coarse-grained, lack coverage, lack accuracy, or do not handle latent failures. We give specifics below and demonstrate some of them experimentally in Section 5.1.

As an example of a coarse-grained mechanism, consider ICMP "destination unreachable" messages, which the network delivers to sources [54]. This signal conflates different failure cases (whether the failure resulted from a problem in the host or network, whether the condition is transient, etc.), requiring that applications react to each failure identically or ignore the notifications altogether.

Other mechanisms do not have good coverage. For example, consider the "connection reset" error in TCP. This signal reports to the application that a remote process has exited—but only if the remote TCP stack and the network are both working.

Other mechanisms have good coverage but lack accuracy. For example, end-to-end timeouts eventually trigger if a failure occurs, but they sometimes trigger prematurely, without any failures.

Some mechanisms do not detect *latent* failures: they report failure only if and when the application tries to use the network. For example, the network generates an ICMP error packet only when a host attempts to send data.[2] As another example, consider timeouts again: they are often set *on* some pending event (e.g., a request issued to a peer). If an application has no such event out-

[2] The `select()` and `epoll()` system calls, which report errors on particular file descriptors, are simply interfaces to this behavior.

condition	occurred?	permanent?	description	example causes
stop	certain	certain	target stopped executing	core dump, machine reboot
unreachability	certain	uncertain	target unreachable	network link down
stop warning	expected; imminent	certain	target may stop executing	disk about to crash
unreachability warning	expected; imminent	uncertain	target may become unreachable	network link close to capacity, CPU overloaded

Figure 1—Conditions reported by Pigeon. These conditions abstract specific failures affecting a remote *target* process and encapsulate two kinds of uncertainty.

standing but later generates one, it must then wait for the timeout interval to expire before learning of the failure.

Falcon [46] detects latent failures and is accurate, but it sacrifices coverage and gives coarse-grained reports. Falcon monitors a remote process with a network of spies deployed at different layers of the system (operating system, application, etc.). If a layer is unresponsive, a spy sometimes kills that layer (e.g., by terminating a virtual machine) so that clients can make progress; this requires network communication so that the Falcon client can request and confirm the kill. As a result, Falcon hangs if there is a network failure. Moreover, Falcon can report applications only as crashed or not crashed.

2.2 Design challenges and principles

As noted in the introduction, there are three top-level challenges in designing Pigeon: identifying what failure details to provide; handling uncertain information safely; and managing a three-way trade-off among coverage, accuracy, and timeliness. At a high level, the root cause of these challenges is the difficulty of determining why a remote process does not respond: is it crashed? or slow? or is the problem in the network? We confront these challenges with the principles below.

Renounce killing. Consider techniques that provide perfect accuracy, such as Falcon [46], watchdogs [1, 27], and virtual synchrony [12]. What would be required for them *not* to hang on network failures? Since their accuracy comes from killing (when they are uncertain), they would have to kill network elements and intentionally create network partitions. This seems like a bad idea. In fact, even targeted killing is not ideal: taking live components offline impairs availability! Pigeon shall not kill.

Provide full coverage. Availability requires that the failure informer report all failures (full coverage). However, two issues result. First, full coverage implies that perfect accuracy is unattainable: if an informer must report all failures (and do so without killing), but is uncertain about whether a failure occurred, then the informer will sometimes report some failures incorrectly. Second, the three-way conflict among coverage, accuracy, and timeliness means that full coverage causes a trade-off between accuracy (already imperfect) and timeliness. Our next two principles address these issues in turn.

Expose uncertainty. How can the failure informer ensure safety, despite occasional mistakes? Our approach is for the failure informer to provide certainty when possible and to flag the reports that may be wrong as uncertain. (This is different from the notion of confidence in failure detectors [34]; see Section 7.) This allows applications to take qualitatively different recovery actions, as stated in the introduction (see also Section 5.2). Note that handling uncertainty is not a burden, as applications do so already when, for example, end-to-end timeouts expire.

Leverage local information. The timeliness-accuracy tradeoff can be improved by local knowledge that reveals the state of components. For example, if a host's cable disconnects from a network switch, the switch quickly learns, and the informer can thus tell the application quickly. For the same accuracy, then, a failure informer with access to lower layers can be more timely, because the local information is visible sooner than if it had to bubble up to higher layers. We borrow the idea of using local information from Falcon [46] (see Section 7).

Design for extensibility. We are not going to get a perfect implementation, so we design for extensibility: Pigeon accommodates add-on modules that provide better information and indicate different kinds of faults, ideally improving the accuracy-timeliness trade-off. These extensions do not require redesigning Pigeon or applications; a key factor in avoiding redesign is exposing failures through an abstraction, versus exposing all details.

3 Design of Pigeon

This section presents the interface exposed by Pigeon (§3.1), describes the guarantees (§3.2), explains how Pigeon can be used (§3.3), describes its architecture (§3.4), and explains errors and their effects (§3.5).

3.1 The failure informer interface

The failure informer interface exposes *conditions* to applications, where each condition abstracts a class of problems in a remote *target process* that all affect the distributed application in similar ways. There are four conditions, shown in Figure 1.

(1) In a *stop*, the target process has stopped executing and lost its volatile state. The problem has already

occurred, and it is certainly permanent. This condition abstracts process crashes, machine reboots, etc.

(2) In an *unreachability*, the target process may be operational, but the client cannot reach it. The problem has already occurred, but it is potentially intermittent. This condition abstracts a timeout due to, say, a network partition or a slow process.

(3) In a *stop warning*, the target process may stop executing soon, as a critical resource is missing or depleted. The problem has not yet occurred, but if it occurs it is permanent. This condition abstracts cases such as a report about an imminent disk failure [33, 53, 63].

(4) In an *unreachability warning*, the target process may become unreachable soon, as an important resource is missing or depleted. The problem has not yet occurred; if it occurs, it is potentially intermittent. This condition abstracts cases such as a network link being nearly saturated or overload in the host CPU of the target process.

The four conditions above reflect a classification based on two types of uncertainty that are useful to applications: whether the problem is certainly permanent (stop vs. unreachability) and whether the problem certainly occurred (actual vs. warning).

The interface also returns *properties*: information specific to the condition, which may help applications recover. A property of all conditions is their expected duration. (Note that a duration estimate does not subsume certainty: certainty-vs-unreachability captures a quality other than duration, and the duration estimate itself is fundamentally uncertain.[3]) We describe how this property is set in Section 4.4. A property of the warning conditions is a bit vector indicating the critical resource(s) responsible for the warning (disk, memory, CPU, network bandwidth, etc.).

Client API. Client applications see the following programmatic interface.

function	description
h = init(target, callback)	request monitoring of target process; returns a handle for use in future operations
uninit(h)	stop monitoring
c = query(h)	get status; returns a list of conditions
res = getProp(h, c, propname)	get condition property value
setTimeout(h, timeout)	set/reset timeout
clearTimeout(h)	cancel timeout

The client calls init() to monitor a target process, named by an IP address and an application identifier in some name space (e.g., port space). The function returns a handle to be used in other functions. The init() function takes as a parameter a callback function, which the implementation calls as new failure conditions emerge.

The query() function returns a (possibly empty) list of active conditions. The getProp() function returns properties, as described above.

The setTimeout() and clearTimeout() functions set/reset and clear end-to-end timeouts. Clients use timeouts as a catch-all: after the client installs a timer, if the client does not cancel or reset it before the timeout period, then the interface reports an unreachability.

[3] In fact, a failure informer can report an unreachability with indefinite (unknown) duration. This is different from a stop, which is permanent.

3.2 Guarantees

We now describe the guarantees provided by Pigeon along three axes: coverage, accuracy, and timeliness. Pigeon provides these guarantees in spite of failures in the network and Pigeon itself, as described in Section 3.5.

Coverage. If the client uses Pigeon's end-to-end timeout, Pigeon guarantees full coverage: if the target process stops responding to the client, then Pigeon reports either a stop or an unreachability condition.

Accuracy. By accuracy, we mean "reported failures are justified" (§1); we address the correctness of duration estimates in Section 5.1. We designed Pigeon not for perfect accuracy in its reports but for accuracy in its certainty: Pigeon knows when it knows, and it knows when it doesn't know. Specifically, if Pigeon reports a stop condition, the application client can safely assume that the target process will not continue; Pigeon returns an unreachability when it cannot confirm that the condition is permanent. When Pigeon reports a warning, it guarantees that a motive exists (a fault occurred) but not that an unreachability or stop will occur.

Timeliness. If a condition occurs, Pigeon reports it as fast as it can. This is a best effort guarantee.

3.3 Using the interface

We now give a general description of how applications might use Pigeon; Section 5.2 considers specific applications (RAMCloud [52], Cassandra [43], lease-based replication [30]). For each of the four conditions, we explain the implications for the application and how it could respond.

Recall that a stop condition indicates that the target process has lost its volatile state and stopped executing permanently; this has a quantitative implication and a qualitative one. Quantitatively, it is safe for the client to initiate recovery immediately. Qualitatively, the client can use simpler recovery procedures: because it gets closure—that is, because it knows that the target process has stopped—it does not have to handle the case that the target process is alive. For example, a stop condition allows the client to simply restart the target on a backup.

By contrast, an unreachability condition implies only

that the target is unreachable; the target process may in fact be operational, or the condition may disappear by itself. This has two implications. First, if the client takes a recovery action, the system may have multiple instances of the target process. Recovering safely therefore requires coordinating with other nodes using mechanisms like Chubby [13], ZooKeeper [35], or Paxos [45], which allow nodes to *agree* on a single master or action. Note that reports of unreachability are still useful—and that using these agreement mechanisms is not overly burdensome—because systems already have the appropriate logic: this is the logic that handles the case that an end-to-end timeout fires without an actual failure.

Second, based on the expected duration of the condition, the application must consider the costs and benefits of just waiting versus starting recovery proactively. Conceptually, each application has an *unavailability threshold* such that if the expected duration of the condition is smaller, the application should wait; otherwise, it should start recovery.

In fact, "eager recovery" can be taken a step further: warnings allow applications to take precautionary actions even without failures. For example, a stop warning could cause an application to bring a stand-by from warm to hot, while an unreachability warning could cause an application to degrade its service.

To illustrate the use of Pigeon concretely, consider a synchronous primary-backup system [4], where the primary serves requests while a backup maintains an up-to-date copy of the primary. The backup can use Pigeon to monitor the primary:

- If Pigeon reports a stop, the backup takes over;
- If Pigeon reports an unreachability, the backup must decide whether to fail over the primary, or instantiate a new replica (either of which requires mechanisms to prevent having multiple primaries), or simply wait. These decisions must weigh the cost of the recovery actions against the expected duration of the condition.
- If Pigeon reports a stop warning, the backup provisions a new replica without failing over the primary.
- Under an unreachability warning, the backup logs the warning so that, if the condition is frequent, operators can better provision the system in the future.

3.4 Architecture of Pigeon

As stated in the introduction, Pigeon works within a single administrative domain: an enterprise, a data center, a campus network, etc. Pigeon's architecture is geared toward extracting and exploiting the information about failures already available inside the system. For example, the failed links in a network collectively yield information about a network partition. To use this information, Pigeon needs mechanisms to (a) sense information inside components, (b) relay information to end-hosts, and (c) interpret information for client applications. These mechanisms are embodied, respectively, in *sensors*, *relays*, and *interpreters* (Figure 2). We describe their abstract function below and their instantiations in our prototype in Section 4.

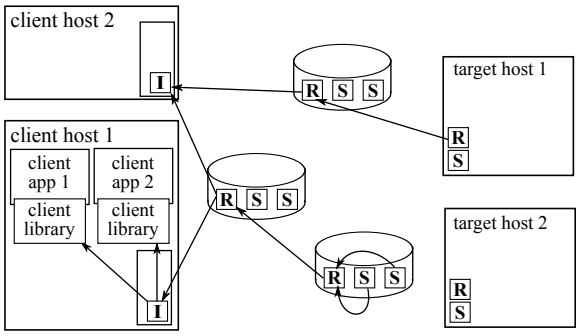


Figure 2—Architecture of Pigeon. Pigeon has sensors (S), relays (R), and interpreters (I). Sensors are component-specific. Sensors and relays are shared by multiple clients and end-hosts; an interpreter is shared by all client applications on its host. The client library presents the client API (§3.1) to applications.

A *sensor* is component-specific and tailored; it is embedded in the component and detects faults in it. A *fault* is a local event, possibly a malfunction, that may contribute to one of the four failure conditions (§3.1). A *critical fault* is one that may lead to a stop condition; a *regular fault*, to an unreachability condition; and an *advisory fault*, to a warning condition. Faults need not cause conditions; they may be masked by recovery mechanisms outside the application (e.g., route convergence).

Relays communicate with sensors and propagate these sensors' fault information to end-hosts. Sensors and relays may be installed for Pigeon or may already exist in the system.

Each end-host has an *interpreter* that receives information about faults from the relays. Interpreters render this information as failure conditions and estimate the expected duration of conditions. Clients interact with interpreters through a *client library*, which implements end-to-end timeouts and the client API (§3.1). Interpreters also determine which sensors are relevant to the client-supplied (IP, port) pair that identifies a target (§4.4).

3.5 Coping with imperfect components

In this section we describe the effect of errors in Pigeon's own components and the network. These errors include crash failures and misjudgments; they do not include Byzantine failures, which Pigeon does not tolerate. Figure 3 summarizes the effect of errors.

Before continuing, we note *non*-effects. First, Pigeon does not compromise on coverage: its coverage derives from the end-to-end timeout, which is implemented in

compromise	cause
coverage	nothing
safety	nothing
timeliness	sensor, relay, or interpreter crashes sensor misses fault interpreter does not report stop or unreachability
accuracy	sensor, relay, or interpreter crashes sensor falsely detects regular or advisory fault interpreter falsely reports unreachability or warning

Figure 3—Effect of errors on Pigeon's guarantees. Errors in duration estimates are covered in Section 5.1.

the client library (linked into the application) and hence shares fate with the client application, despite failures elsewhere. Second, Pigeon is designed to not compromise safety; while *inaccuracy* is possible under Pigeon, the only threat to *safety* is a report of a stop that did not happen (§3.2), which Pigeon is designed to avoid (§4).

If a sensor, relay, or interpreter crashes or is disconnected from the network, Pigeon loses access to local information, which affects accuracy and timeliness (§2.2). Loss of local information also causes missed opportunities to report some failures as stop conditions (e.g., remote process exit) rather than an unreachability condition triggered by the end-to-end timeout.

If a sensor does not detect a fault, then Pigeon may need to rely on the end-to-end timeout, compromising timeliness. If a sensor falsely detects a regular fault, then Pigeon may misreport an unreachability condition. This error in turn compromises accuracy (potentially causing an unwarranted application recovery action) but not safety (see above). The effect when a sensor falsely detects an advisory fault is similar (misreports of warning conditions).

If the interpreter crashes or fails to report a condition, then Pigeon relies on the end-to-end timeout, again compromising timeliness. If the interpreter misreports an unreachability or warning, Pigeon compromises accuracy but not safety (see above). Errors in the interpreter's duration estimates are covered in Section 5.1.

We have designed Pigeon to be extensible, so new components can reduce the errors above. However, Pigeon's current components, which we describe next, already yield considerable benefits.

4 Prototype of Pigeon

We describe our target environment (§4.1), and the implementations of sensors (§4.2), relays (§4.3), and the interpreter (§4.4) used in our prototype. The prototype borrows many low-level mechanisms from prior work, as we will note, but the synthesis is new (if unsurprising).

4.1 Target environment

Our prototype targets networks that use link-state routing protocols, which are common in data centers and enterprises [31, 39]. Currently, the prototype assumes the Open Shortest Path First (OSPF) protocol [51] with a single OSPF *area* or routing zone. This assumption may raise scalability questions, which we address in Section 5.3. We discuss multi-area routing and layer 2 networks in Section 6.

We assume a single administrative domain, where an operator can tune and install our code in applications and routers; this tuning is required at deployment, not during ongoing operation.

4.2 Sensors

Sensors must detect faults quickly and confirm critical faults; the latter requirement ensures that Pigeon does not incorrectly report stops. The architecture accommodates pluggable sensors, and our prototype includes four types: a *process sensor* and an *embedded sensor* at end-hosts, and a *router sensor* and an *OSPF sensor* in routers. For each type, we describe the faults that it detects, how it detects them, and how it confirms critical faults. Faults are denoted as F-⟨type⟩ (critical ones noted in parentheses).

Process sensor. This sensor runs at end-hosts. When a monitored application starts up, it connects to its local process sensor over a UNIX domain socket. The process sensor resembles Falcon's application spy [46], but it does not kill. The sensor detects three faults:

F-exit (critical). The target process is no longer in the OS process table and has lost its volatile state, but the OS remains operational. This fault can be caused by a graceful exit, a software bug (e.g., segmentation fault), or an exogenous event (e.g., the process was killed by the out-of-memory killer on Linux). To detect this fault, the sensor monitors its connection to the target processes. When a connection is closed, the sensor checks the process table every $T_{proc\text{-}check}$ time units; after confirming the target process is absent, it reports F-exit. Our prototype sets $T_{proc\text{-}check}$ to 5 ms, a value small enough to produce a fast report, but not so small as to clog the CPU.

F-suspect-stop. The target process is in the process table but is not responding to local probes. This fault can be due, for example, to a bug that causes a deadlock in the target process. To detect this fault, the sensor queries the monitored process every $T_{app\text{-}check}$ time units. If the target process reports a problem or times out after $T_{app\text{-}resp}$ time units, the sensor declares the fault. Our prototype sets $T_{app\text{-}check}$ to 100 ms of real time and $T_{app\text{-}resp}$ to 100 ms of CPU time of the monitored application (the same values are justified in Falcon [46, §4]).

F-disk-vulnerable. A disk used by the target process has failed or is vulnerable to failure (based on vendor-specific reporting data, e.g., SMART [63]). To detect this fault, Pigeon checks the end-host's SMART data every $T_{disk\text{-}check}$ time units, which our prototype sets to 500 ms.

Embedded sensor. The next sensor is logic embedded in the end-host operating systems. This sensor resembles Falcon's OS-layer spy but has additional logic to confirm critical faults without killing. It detects three faults:

F-host-reboot (critical). The OS of the target process is rebooting. The embedded sensor reports this fault during the shutdown that precedes a reboot but only after all of the processes monitored by Pigeon have exited (the waiting prevents falsely reporting a stop condition).

F-host-shutdown (critical). The OS of the target process is shutting down. The sensor uses the same mechanism as for F-host-reboot.

F-suspect-stop. The OS of the target process is no longer scheduling a high priority process that increments a counter in kernel memory every T_{inc} time units. The sensor detects a fault by checking that the counter has incremented at least once every $T_{inc\text{-}check}$ time units. Our prototype sets T_{inc} and $T_{inc\text{-}check}$ to 1 ms and 100 ms, respectively, providing fast detection of failures with negligible CPU cost (we borrow these settings from Falcon).

Router sensor. A process on the router runs as a sensor that detects two faults:

F-suspect-stop. The end-host is no longer responding to network probes. This fault can occur, for example, because of a power failure or an OS bug. The router sensor detects this fault by running a keep-alive protocol with any attached end-hosts. (This keep-alive protocol is borrowed from Falcon.)

F-link-util. A network link has high utilization. Our prototype checks the utilization of the router's links every T_{util} time units and detects a fault if utilization exceeds a fraction F_{bw} of the link bandwidth. Our prototype sets F_{bw} to 63% (which we measured to be the lowest utilization at which a router starts to drop traffic) and T_{util} to 1 second (which corresponds to the maximum rate at which this fault can be reported; see Section 4.3).

OSPF Sensor. A router's OSPF logic acts as a sensor that detects two faults:

F-link. A link in the network has gone down. The routers in our environment detect link failures using Bidirectional-Forwarding Detection (BFD) [38].

F-router-reboot. A network router is about to reboot. The sensor detects this fault because the operating system notifies it that the router is about to reboot.

4.3 Relays

The prototype uses three kinds of relays: one at end-hosts, called a *host relay*, and two at routers, called a *router relay* and an *OSPF relay*. Relays may be faulty, as discussed in Section 3.5.

Host relay. This relay communicates faults detected by the process sensor, and it runs in the same process as the process sensor. When a client begins monitoring a target process, the client's interpreter registers a callback at the target's host relay. The host relay invokes this callback whenever the process sensor detects a fault. Callbacks improve timeliness, as the interpreter learns about faults soon after they happen; this technique is used elsewhere [20, 36, 46].

Router relay. This relay communicates the F-suspect-stop fault detected by the router sensor, as well as all faults detected by the embedded sensors. The relay runs in the same process as the router sensor, and it uses the same callback protocol as the host relay.

OSPF relay. This relay uses OSPF's link-state routing protocol to communicate information about links. Under this protocol, routers generate information about their links in *Link-State Advertisements* (LSAs) and propagate LSAs to other routers using OSPF's flooding mechanism. For link failures (F-link), the OSPF relay uses normal LSAs, and for graceful shutdowns (F-router-reboot), the relay uses LSAs with infinite distance [57]. To announce overloaded links (F-link-util), the router relay uses *opaque LSAs* [10], which are LSAs that carry application-specific information.

Using the network to announce overload and failures might compound problems, so we rate-limit opaque LSAs to R_{opaque}, which our prototype sets to 1 per second (the highest rate at which routers should accept LSAs [10]). Similarly, a buggy client could deplete the resources of this relay (and the router relay), since they are shared; mitigating such behavior is outside our current prototype's scope, but standard techniques should apply (rate-limiting, etc.). Note that the concern is buggy clients, not malicious ones, because Pigeon targets a single administrative domain (§4.1).

4.4 The interpreter

The interpreter gathers information about faults and outputs the failure conditions of §3.1. The interpreter must (1) determine which sensors correspond to the client-specified target process, (2) determine if a condition is implied by a fault, (3) estimate the condition's duration, (4) report the condition to the application via the client library, and (5) never falsely report a stop condition. We discuss these duties in turn.

(1) The interpreter determines which sensors are relevant to a target process by using knowledge of the network topology, the location of sensors, and the location of the client and target processes.

(2) The interpreter must not report every fault as a condition; for example, a failed link that is not on the client's path to the target does not cause an unreachability condition. If the interpreter cannot determine the effect of a fault from failure information alone, it uses *hints*. For

example, if a link becomes loaded along one of multiple paths to the target process, the interpreter sends an ICMP Echo Request with the Explicit Congestion Notification (ECN) option [56] set, to determine if the client's current path is affected. The router sensors intercept these packets, and, if a link is loaded, mark them with the Congestion Encountered (CE) bits. If the interpreter receives an Echo Reply with these bits set, or times out after $T_{probe\text{-}to}$ time units, the interpreter reports an unreachability warning; in this warning, the network is marked as the critical resource (§3.1). Our implementation sets $T_{probe\text{-}to}$ to 50 ms.[4] The interpreter uses a similar hint (a network probe packet) to determine the effect of link failures.

The interpreter determines which paths are available to clients by passively participating in OSPF, a technique used elsewhere [36, 59, 60]. For detecting link failures, this technique adds little overhead to the network. However, detecting link utilization has additional overhead (because it generates extra LSAs), and OSPF itself has some cost. We evaluate these costs in Section 5.3.

(3) As mentioned earlier, the interpreter estimates the duration of some unreachability conditions. Currently, these durations are hard-coded based on our testbed measurements, which we describe next; a better approach is to estimate duration using on-line statistical learning.

Our prototype estimates the duration of unreachability conditions as follows. If a link fails or a router reboots along the current path from the client to the target process, but there are alternate working paths, the interpreter reports a duration of $T_{new\text{-}path\text{-}delay}$—the average time that the network takes to find and install the new path. If a router reboots and there are no working paths from the client to the target process, the client must wait for the router to reboot, so the interpreter reports a duration of $T_{router\text{-}reboot}$—the average time that the router takes to reboot. The interpreter reports all other conditions as having an indefinite duration.

In our testbed, we set $T_{new\text{-}path\text{-}delay}$ and $T_{router\text{-}reboot}$ to 2.8 seconds and 66 seconds, respectively. We determine these values by measuring the unavailability caused by a fault, as observed by a host pinging another every 50 ms. In each experiment, we inject a link failure or router reboot, and report the failure's duration as the gap in ping replies observed by the end-host. We repeat this experiment 50 times for each fault. The means are as reported; the standard deviations are 27 ms and 2.5 seconds, respectively, for the two conditions.

(4) The interpreter reports all conditions (and their expected duration) to the client library; the interpreter also informs the client library if a condition clears or changes expected duration. The client library in turn calls back the client, and also exposes active conditions via the query() function (§3.1).

(5) To avoid reporting false stop conditions, the interpreter reports a stop only for the critical faults (F-exit, etc.), which sensors always confirm (by design).

[4] We validate this timeout by running an experiment where one host sends an ICMP Echo Request to another host for 10,000 iterations in a closed loop. We observe a response latency (which includes round-trip time and packet processing time) of 760 μs (standard deviation 96 μs) and a maximum of 1.2 ms, well below the timeout value.

Compared to existing failure reporting services, Pigeon improves, either in coverage, accuracy, timeliness, or quality	§5.1
Pigeon's richer information enables applications to react quickly or prevent costly recoveries	§5.2
Pigeon uses negligible CPU and moderate network bandwidth	§5.3

Figure 4—Summary of main evaluation results.

what problem is modeled?	how is the fault injected?
process crash	segmentation fault
host reboot	issue reboot at host
link failure (backup paths exist)	disable router port
link failures (partition)	disable multiple router ports
router reboot (disrupts all paths)	issue reboot at edge router
network load	flood network path with burst
disk failure	change SMART attributes [63]

Figure 5—Panel of modeled faults. The three groups should generate stop, unreachability, and warning reports, respectively.

5 Experimental evaluation

We evaluate Pigeon by assessing its reports (§5.1), its benefit to applications (§5.2), and its costs (§5.3). Figure 4 summarizes the results.

Fully assessing Pigeon's benefit would require running Pigeon against real-world failure data. We do not have that data, and gathering it would be a paper in its own right [28]. Instead, we consider several real-world applications and failure scenarios, and show Pigeon's benefit for these instances.

Specifically, our evaluation compares our prototype to a set of baselines, in a test network, under synthetic faults. The three *baselines* in our experiments are:

1. End-to-end timeouts, set aggressively (200 ms timeout on a ping sent every 250 ms) and to more usual values (10 second timeout; ping every 5 seconds).
2. Falcon, with and without killing to confirm failure. We call the version without killing Falcon-NoKill.
3. A set of Linux system calls (§2.1): `send()` invoked every 250 ms, `recv()`, and `epoll()`, with and without error queues.

Our test network has 16 routers and 3 physical hosts, each multiplexing up to 4 virtual machines (VMs).[5] Our

[5] We do not expect much loss of fidelity in network performance from using VMs. The peak throughput achieved by a benchmark tool, net-

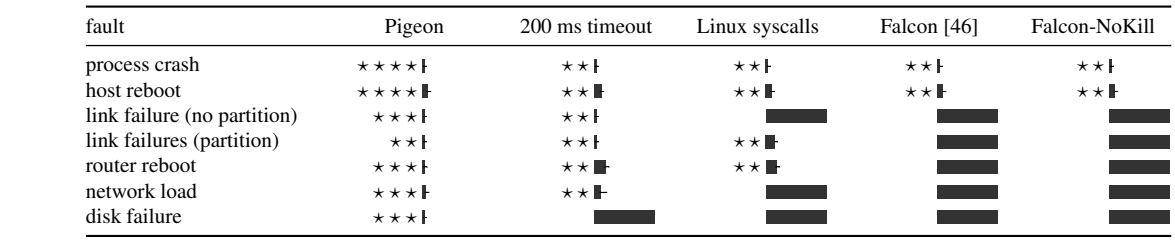
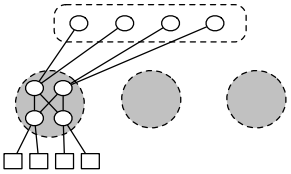


Figure 6—Pigeon compared to baseline failure reporters under our fault panel. **More stars and smaller bars are better.** Stars indicate the quality of a report; bars indicate the detection time. A maximum of four stars are awarded for detecting a failure, giving a certain report, giving more information than just crashed-or-not (e.g., indicating the cause as network load), and for not killing. Bar length and error bars depict mean detection time and standard deviation. These quantities are scaled; maximum is 30 seconds (long bars), which means "not covered". For the faults in our panel, Pigeon has higher quality, lower detection time, or both.

testbed looks like this:



It comprises three *pods* (gray circles), consisting of four routers (white circles) and hosts (white squares). This is a *fat-tree* topology [3], which we use to model a data center. Note that our operating assumptions are data centers, fat-tree, and OSPF; these assumptions are compatible, as data centers use OSPF.[6] Our topology has the same size as the one evaluated by Al-Fares et al. (minus one pod), albeit with different hardware [3].

Our routers are ASUS RT-N16s that run DD-WRT (basically Linux) [25], and use the Quagga networking suite [55] patched to detect link failure with BFD [38]. Our hypervisors run on three Dell PowerEdge T310s, each with a quad-core Intel Xeon 2.4 GHz processor, 4 GB of RAM, and ten Gigabit Ethernet ports (four of which are designated for VMs). The VMs are guests of QEMU v1.1 and the KVM extensions of the Linux 3.4.9-gentoo kernel. The guests run 64-bit Linux (2.6.34-gentoo-r6) and have either 768 MB of memory (labeled *small*) or 1536 MB of memory (*large*). Each VM attaches to the network using the host's Intel 82574L NIC, which it accesses via PCI passthrough.

Figure 5 lists the panel of faults in our experiments. Although the *faults* are synthetic, the resulting *failures* model a class of actual problems.

5.1 How well does Pigeon do its job?

In this section, we first evaluate Pigeon's reports and then the effect of duration estimation error.

Multi-dimensional study. There are many competing requirements in failure reporting; the challenge is *not* to meet any one of them but rather to meet all of them. Thus, we perform a multi-dimensional study of Pigeon and the baselines.

Quantitatively, we investigate timeliness: for each pair of failure reporter and fault, we perform 10 runs in which a client process on a (small) VM monitors a target process on another (small) VM in the same pod. We record the detection time as the delay between when the apparatus issues an RPC (to fault injection modules on the routers and hosts) and when the client receives an error report; if no report is received within 30 seconds, we record "not covered". Qualitatively, we develop a rating system of failure reporting features: certainty, ability to give warnings, etc.

Figure 6 depicts the comparison. Pigeon's reports are generally of higher quality than those of the baselines; for instance, Falcon offers certainty, but it kills to do so. And none of the baselines gives proactive warnings, as Pigeon does for the final two faults in the panel. In Section 5.2, we investigate how these qualitative differences translate into benefits for the application.

Pigeon's reports are timely. For process crashes, single link failure, partition, and router reboot, the mean detection times are 10 ms, 710 ms, 660 ms, and 690ms. For host reboots, Pigeon has a mean detection time of 1.9 seconds. (Detecting host reboot takes longer because we measure from when the reboot command is issued, and there is delay between then and when the reboot affects processes.)

Pigeon has full coverage, at least in our experiments. Finally, we come to accuracy (recall that Pigeon has to balance coverage, timeliness, and accuracy). In our experiments, Pigeon is accurate: we never observe Pigeon incorrectly reporting a fault that has not occurred (a production deployment would presumably see some false reports and could adjust its parameters should such reports become problematic; see Section 4). Next, we consider the effect of duration estimation error in Pigeon's reports.

Duration estimation error. To understand the effect of duration estimation error, we compare our prototype to an *ideal failure informer* that predicts the exact duration of a failure condition. Specifically, we measure the ad-

perf [2], is the same for a virtual and physical machine in our testbed, and in our experiments, VMs do not contend for physical resources.

[6] A non-assumption is using layer 3: there are data center architectures, based on fat-tree variants, that use OSPF at layer 2 [31].

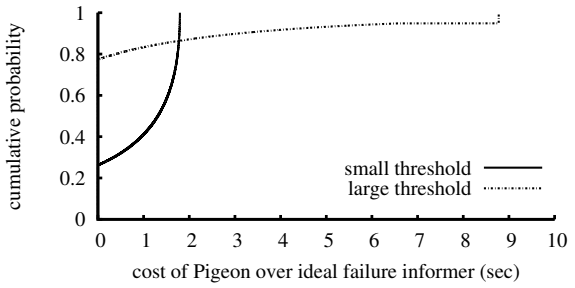


Figure 7—CDF of Pigeon's cost over the ideal failure informer for two sample applications, with availability thresholds (§3.3) smaller and larger than Pigeon's duration estimate.

ditional unavailability that Pigeon causes in two applications: one that always recovers when using Pigeon because its unavailability threshold (§3.3) is smaller than Pigeon's estimate (which is static; see Section 4.4), and one that always waits (because its threshold is higher).

We perform a simulation; we sample failure durations from a Weibull distribution (shape 0.5, scale 1.0), which is heavy-tailed and intended to stress the prototype's static estimate by "spreading out" the range of actual failures. For each sample, we record the *cost*, defined as the additional unavailability of the application when it uses Pigeon versus when it uses the ideal. We model the application's recovery duration and availability threshold as equal to each other.

Figure 7 depicts the results. For the small threshold, Pigeon matches the ideal for fewer than 30% of the samples because a significant fraction of the actual durations are very close to zero. Since this application always recovers with Pigeon, it frequently incurs (unnecessary) unavailability from recovery: waiting out these short failures would have resulted in less unavailability. For the large threshold, Pigeon matches the ideal for almost 80% of the samples but sometimes does much worse, since it waits on a long tail of failure durations. However, both applications' costs are capped, owing to their backstop timeouts. Additionally, we find that these simulated applications incur lower costs from using Pigeon compared to choosing "recover" or "wait" uniformly at random.

5.2 Does Pigeon benefit applications?

We consider three case study applications that use Pigeon differently: RAMCloud [52], Cassandra [43], and lease-based replication [30]. For each, we consider the unmodified system, the system modified to use Pigeon, and the system modified to use one or more baselines.

RAMCloud [52]. RAMCloud is a storage system that stores data in DRAM at a set of *master servers*, which process client requests. RAMCloud replicates data on the disks of multiple *backup servers*, for durability. To reduce unavailability after a master server fails, a *coordinator* manages recovery to reconstruct data from the backups quickly. There are two notable aspects of RAMCloud for our purposes. First, although recovery is fast, it is expensive (it draws data from across the system, and it ejects the server, reducing capacity). Second, RAMCloud has an aggressive timeout: it detects failures by periodically pinging other servers at random and then timing out after 200 ms.

	RAMCloud using		
fault	timeout	Falcon [46]	Pigeon
process crash	2.7s, eject	2.1s, eject	1.9s, eject
host reboot	2.6s, eject	1.8s, eject	1.9s, eject
link failure (no partition)	2.8s, eject	2.6s, wait	2.6s, wait
link failures (partition)	2.6s, eject	∞, wait	2.6s, eject
router reboot	2.6s, eject	∞, wait	1.7s, eject
network load	∞, eject	0.5s, wait	0.5s, wait

Figure 8—Mean unavailability observed by a RAMCloud client when RAMCloud uses different detection mechansims (standard deviations are within 15% of means). We also note whether RAMCloud ejects a server or waits for the fault to clear. Pigeon is roughly as timely as highly aggressive timeouts but saves RAMCloud the cost of recovery sometimes (under link failure (no partition) and network load faults). Falcon [46] hangs on network failures, so RAMCloud+Falcon does too (represented with ∞). Using timeouts, RAMCloud sometimes hangs if network load triggers multiple recoveries.

Thus, we expect that unmodified RAMCloud recovers more often than needed, and that Pigeon could help it begin recovery quickly or avoid recovering; we also expect that Pigeon can offer this benefit while providing full coverage and timely information. To investigate, we modify RAMCloud servers to use Pigeon and Falcon (with long backstop timeouts that do not fire in these experiments). We run a RAMCloud cluster on six large VMs (one client, five servers; two VMs in each pod), where each server stores 20MB of data. This configuration allows RAMCloud to recover quickly on our testbed, at the cost of ejecting a server. For each injected fault, we perform 10 iterations and measure the gap in response time that is seen by a client querying in a closed loop.

Figure 8 depicts the results. Pigeon is roughly as timely as very aggressive timeouts, deriving its timeliness from sensors. Pigeon also enables RAMCloud to forgo recovery when possible. For instance, RAMCloud waits under network load when it receives a warning from Pigeon. Under a link failure, RAMCloud receives an unreachability condition with a short duration (equal to the network convergence time), so it waits. By contrast, under router reboot, RAMCloud receives an unreachability condition with a long duration (see Section 4.4), so it recovers.

Cassandra [43]. Cassandra [43] is a distributed key-value storage system used broadly (e.g., at Netflix, Cisco,

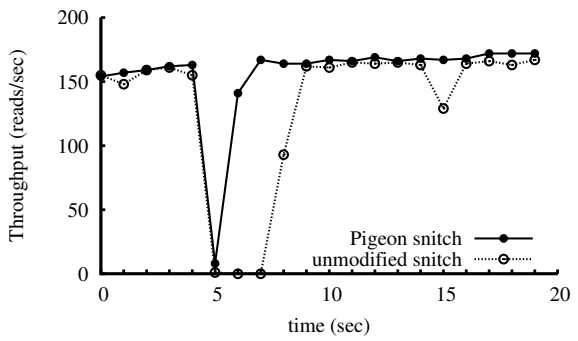


Figure 9—Cassandra's read throughput with and without Pigeon, after a network link fails 5 seconds into the run, temporarily disrupting a single server. Using Pigeon, the Cassandra snitch avoids using an unreachable replica; without Pigeon, Cassandra waits for the server to become reachable again. This example is representative: in our experiments, clients observed a mean unavailability of 1 second ($\sigma < 0.1$) using Pigeon and 2.2 seconds ($\sigma = 1.3$) using the unmodified snitch.

and Reddit [16]). Cassandra servers read data from a primary replica and request *digests* from the other replicas. Thus, the choice of primary is important: if the primary has a problem, the server blocks until the problem is solved or the request times out. A server chooses as its primary the replica with the lowest expected request latency, as reported by an *endpoint snitch*.

We expect that Pigeon could help a snitch make better server selections. To measure this benefit, we run a client in a closed loop, inject two faults (network load and link failure) at a server in a five-server cluster (using large VMs), and measure the throughput.

Under network load (not depicted), the unmodified snitch and the Pigeon snitch offer comparable (significant) benefit over no snitch, as the unmodified snitch's decisions are based on latencies—but only if the network is working. This brings us to Figure 9, which depicts the link failure case: here, Pigeon's report to the snitch allows the server to quickly choose a better primary, resulting in higher throughput. Compare to RAMCloud: Pigeon lets Cassandra act more quickly than it otherwise would (because Pigeon reports the case and because switching is cheap), whereas this same report lets RAMCloud wait when it would otherwise act (see above).

Lease-based replication [30]. A common approach to replication is to use a *lease server* [13, 30], which grants a lease to a *master* replica, which in turn handles client requests, forwarding them to *backups*. If a backup detects or suspects a failure, it tries to become the master, by requesting a lease from the lease server. However, this process is delayed by the time remaining on the lease.

We expect that Pigeon's stop reports would be particularly useful here: they report that a lease holder has crashed with certainty, which allows the system to break the lease, increasing system availability.[7] To investigate, we build a demo replication application and lease server, which offers 10-second leases, and run it with and without Pigeon. We run a client (10 iterations) that issues queries in a closed loop, measuring the response gap seen by the client after we inject a process crash at the master.

The results are unsurprising (but encouraging): the response gap measured at the client averages 2.7 seconds (standard deviation 0.4 seconds) when using Pigeon, versus 6.1 seconds (standard deviation 2.5 seconds) using unmodified lease expiration.

Which applications do not gain from Pigeon? We considered simple designs for many applications; Pigeon usually provides a benefit but sometimes not. For example, a DNS client can use Pigeon to monitor its DNS server and quickly failover to a backup server when there is a problem. However, because the client's recovery is lightweight (retry the request), there is little benefit over using short end-to-end timeouts, since the cost of inaccuracy is low. Some applications do not make use of *any* information about failures; such applications likewise do not gain from Pigeon. For example, NFS (on Linux) has a *hard-mount mode*, in which the NFS client blocks until it can communicate with its NFS server; this NFS client does not expose failures or act on them. However, such applications are not our target since they consciously renounce availability.

5.3 What are Pigeon's costs?

Implementation costs. Pigeon has 5.4K lines of C++ and Java. Sensors are compact, and the system is easy to extend (e.g., the disk failure logic required only 34 lines). Integrating Pigeon into applications is easy: it required 68 lines for RAMCloud and 414 lines for Cassandra.

CPU and network overheads. Figure 10 shows the resource costs of Pigeon. CPU use is small; the main cost is a high-priority process in the embedded sensor, which periodically increments a shared counter (§4.2). Pigeon's network overheads come from OSPF LSAs to hosts.

Scalability. The main limiting factor is bandwidth to propagate failure data; this overhead is inherited from OSPF, which generates a number of LSAs proportional to the number of router-to-router links in the network. And this many LSAs are reasonable for networks with thousands of routers and tens of thousands of hosts. Specifically, we estimate that in a 48-port fat-tree topology with 2880 routers and 27,648 end-hosts [3], OSPF would use less than 11.8 Mbps of bisection bandwidth (or 1.1% of 1 Gbps capacity), which is consistent with our smaller-scale measurements. Larger networks would

[7] Note that Falcon would also enable such lease-breaking, but Falcon is incompatible with the availability requirement: if the problem is in the network, a query to Falcon literally hangs.

component (§4)	detecting network load	idle
CPU *used at end-hosts*		
process sensor/host relay	0.1%	0.0%
embedded sensor	3.0%	0.0%
interpreter	0.0%	0.0%
CPU *used at routers*		
router sensor/relay	0.2%	0.0%
OSPF sensor/relay	0.1%	0.0%
bandwidth used		
at each end-host	2.3 kbps	0 bps
at each router	3.4 kbps	1.3 kbps

Figure 10—Resource overheads of our Pigeon implementation.

presumably use multiple areas; we briefly discuss extending Pigeon to that setting in the next section.

6 Discussion, limitations, and future work

We now consider assumptions and limitations of the failure informer abstraction (§3.1–§3.2), the Pigeon architecture (§3.4), and our prototype implementation (§4).

The abstraction. How do we know if we got the abstraction right? As with any abstraction, this one is based on generalizing from specific difficult cases, on judgment, and on use cases. It is hard to prove that an abstraction is optimal (but ours is better than at least our own previous attempts). A critique is that an implementation of the abstraction is permitted to return spurious "uncertain" reports. However, uncertainty is fundamental and hence some wrong answers are inevitable (§2.2); thus, this critique is really a requirement that the implementation have few false positives (§5.1).

The architecture. Our architecture assumes a single administrative domain. This scenario has value (many data centers satisfy this assumption), but extending to a federated context may be worthwhile. However, this requires additional research; prior work gives a starting point [5, 6, 8, 61, 68].

The prototype. Our prototype assumes OSPF, runs on layer 3, and monitors only end-hosts and routers (not middleboxes). We could extend our prototype to other routing protocols, by implementing appropriate relays and sensors (§4.2–§4.3). We could also extend to layer-2 networks, either with OSPF (some layer-2 architectures run OSPF for routing [31]), or without; in the latter case, the prototype would need different sensors and relays. Another extension is to monitor middleboxes using additional types of sensors. Neither our current prototype nor these extensions requires structural network changes. (The logic for sensors and relays is small and runs in software, on a router's or switch's control processor.)

We estimated our prototype's scalability in Section 5.3: it ought to scale to tens of thousands of hosts in a single area, with the limit coming from OSPF itself. OSPF can scale to more hosts, by using multiple areas; we could extend Pigeon to this case using additional sensors and relays at area borders to address what would otherwise be a loss of accuracy (since areas are opaque to each other). We leave this for future work.

7 Related work

Pigeon borrows low-level mechanisms from prior work in network monitoring and failure handling. We describe these two areas, and also present an extended comparison with Falcon [46], which is Pigeon's closest relative.

Network monitoring and intelligence. Many works in network monitoring [7, 9, 23, 26, 29, 40, 41, 67, 69] are complementary to Pigeon. Broadly speaking, these works extract intelligence from network elements to aid diagnosis, and Pigeon could use these techniques. Indeed, Pigeon's OSPF monitoring technique is borrowed from Shaikh et al. [59, 60] (see Section 4.4). However, the goal of these works is to help network operators perform diagnosis while Pigeon's is to provide an online failure reporting abstraction to distributed applications.

Providing a comprehensive service to distributed applications, using global information about the state of a network, is the goal of *information planes* [19, 65]. Works in this area include the Knowledge Plane [22], Sophia [58, 65] (which provides a distributed computational model for queries), iPlane [49, 50] (which helps end-host applications choose servers, peers, or relays, based on link latency, link loss, link capacity, etc.), and NetQuery [61] (which instantiates a Knowledge Plane under adversarial assumptions). These works are more flexible than Pigeon (they usually expose an interface to arbitrary queries), while Pigeon is more focused: its goal is to report failure conditions to applications, a capability that these papers do not discuss.

More targeted works include Meridian [66] (a node and path selection service), King [32] (a latency estimation service), and Network Exception Handlers [36] (NEHs), which proactively delivers information from the network to the end-host operating system, so end-hosts can participate in traffic engineering. The goals of these systems are different from Pigeon's goal of exposing failures. But again, Pigeon could be extended to use similar techniques, and in fact, Pigeon's callback-based architecture is reminiscent of the delivery mechanism in NEH.

While there are works that do report network failures and errors to end-hosts [5, 42, 64], they do not provide a comprehensive abstraction or full coverage, in contrast to Pigeon's goal. For example, Packet Obituaries [5] (POs) proposes that each dropped packet should generate a report about which AS dropped it. Their credo ("keep the host informed!") is similar to ours, and information about POs would be useful for Pigeon, but POs do not obviate Pigeon. First, under POs, the network generates reports

	Falcon [46]	Pigeon
interface	failure detector (crashes)	failure informer (§3.1–§3.2)
accuracy	always accurate	usually accurate
timeliness	fast	fast
domain	host failures	host+network failures
coverage	incomplete	full
blocks	yes	no
kills	yes	no

Figure 11—Pigeon compared to its most closely related system, Falcon [46]. Falcon has better accuracy, which simplifies the layers over it, but Pigeon is superior in the other respects and in particular leads to higher availability.

proactively but only when the host sends a packet, so this mechanism has the limitation discussed in Section 2.1, of allowing latent failures to persist. Second, POs provide low-level information about individual packets, in contrast to Pigeon's higher-level goal. Third, POs do not provide information about host failures.

Failure recovery and detection. Handling failures requires *recovery* and *detection*. *Host* failure recovery (see [24] and citations therein) is complementary to our work. (From our vantage, strategies such as microreboot [14, 15] are about masking and containing faults; for us, recovery is about what to do when faults cannot be masked.) *Networks*, of course, are designed for recovery, but there are techniques for making them even more robust: Failure Carrying Packets [44] and SafeGuard [47] mask failures by carrying control plane information inside data packets, and in Data-Driven Connectivity [48], data plane packets trigger limited routing state changes. Though these works are orthogonal to ours, an open question is whether applications can benefit from knowing that fault masking is underway.

The other aspect of handling failures is *detection*. Chandra and Toueg [17] gave a theory of failure detection, in the context of a client monitoring a remote process. Since then, a number of failure detectors (FDs) based on end-to-end timeouts have been proposed, including by Bertier et al. [11], Chen et al. [18], and So and Sirer [62]. The ϕ-accrual FD, by Hayashibara et al. [34], extends the FD interface with a measure of *confidence*. This notion of confidence contrasts with Pigeon's notion of "certain crash": the confidence is probabilistic, so even when the ϕ-accrual failure detector reports a crash with high confidence, the monitored process may be up. The failure detection literature, particularly the Falcon failure detector [46], influenced the design of Pigeon; we compare the two systems immediately below.

Pigeon vs. Falcon [46]. Falcon observed the power of low-level information, and Pigeon borrows this observation, but the two have different goals, different properties, and different designs. Figure 11 shows the differences. Falcon is an accurate failure detector [17]—an existing abstraction that reports crash or up; by contrast, Pigeon presents a new abstraction (the failure informer) that exposes more information but with less accuracy. Furthermore, Falcon does not have full coverage of hosts or any coverage of the network; in the non-covered cases, it hangs. In terms of design, Falcon (a) uses low-level information only from hosts, (b) relies on the layered structure of end-host system software, and (c) relies on killing. Pigeon faces a bigger problem (network *and* host failures, and a richer interface), in a landscape that does not admit a layered structure or a license to kill. Thus, Pigeon needs a different design, one that has intelligence from the network and better local knowledge. Furthermore, there is a philosophical distinction in the knowledge provided: Falcon reports the things that it knows it knows, while Pigeon *in addition* gives timely reports of the things that it knows it doesn't know, and eventual reports of the things that it does not know it does not know.

8 Summary and conclusion

> The Internet is transparent to success but opaque to failure [5].

Pigeon's top-level contributions are architectural: a thesis that applications should get information about failures, and a proposal to encapsulate that information in a new abstraction that conveys the degree of certainty. Of course, there is much about Pigeon to object to: its ultimate goal (better availability) is shared by all, its design is unsurprising, its mechanisms are borrowed, and its implementation is limited. Nevertheless, this derivative system in fact enables higher application availability, and it does so by enabling new behavior and functionality in applications. Specifically, applications can use the information provided by Pigeon to take the most appropriate action for the failure at hand: to initiate recovery more quickly, to execute a simpler recovery strategy, to recover proactively, or to simply wait it out by not recovering yet. As demonstrated in our experimental evaluation, this freedom leads to qualitatively and quantitatively better behavior, for a modest price in resources. Pigeon, then, is like its namesake: in the wrong environment, it is a homely nuisance; in the right one, it is a key tool with surprisingly powerful functionality.

Acknowledgments

This paper was improved by the helpful comments of Lorenzo Alvisi, Sebastian Angel, Mahesh Balakrishnan, Russ Cox, Alan Dunn, James Grimmelmann, Rodrigo Rodrigues, Srinath Setty, Scott Shenker, and Edmund L. Wong. We thank the anonymous reviewers, and our shepherd Katerina Argyraki, for their suggestions. This research was supported in part by AFOSR grant FA9550-10-1-0073 and NSF grants 1055057 and 1040083.

References

[1] Linux-HA, High-Availability software for Linux. `http://www.linux-ha.org`.

[2] Netperf, the network performance benchmark. `www.netperf.org`.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM*, pages 63–74, Aug. 2008.

[4] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *International Conference on Software Engineering (ICSE)*, pages 562–570, 1976.

[5] K. Argyraki, P. Maniatis, D. Cheriton, and S. Shenker. Providing packet obituaries. In *ACM Workshop on Hot Topics in Networks (HotNets)*, Nov. 2004.

[6] K. Argyraki, P. Maniatis, and A. Singla. Verifiable network-performance measurements. In *ACM Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, Dec. 2010.

[7] H. Ballani and P. Francis. Fault management using the CONMan abstraction. In *INFOCOM*, Apr. 2009.

[8] B. Barak, S. Goldberg, and D. Xiao. Protocols and lower bounds for failure localization in the Internet. In *EUROCRYPT*, Apr. 2008.

[9] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 335–348, Apr. 2009.

[10] L. Berger, I. Bryskin, A. Zinin, and R. Coltun. The OSPF opaque LSA option. RFC 5250, Network Working Group, July 2008.

[11] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *International Conference on Dependable Systems and Networks (DSN)*, pages 354–363, June 2002.

[12] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 123–138, Nov. 1987.

[13] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–350, Dec. 2006.

[14] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive microreboots: A soft-state system case study. *Performance Evaluation Journal*, 56(1–4):213–248, Mar. 2004.

[15] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot—a technique for cheap recovery. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–44, Dec. 2004.

[16] The Apache Cassandra Project. `http://cassandra.apache.org/`.

[17] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

[18] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(5):561–580, May 2002.

[19] B. Chun, J. M. Hellerstein, R. Huebsch, P. Maniatis, and T. Roscoe. Design considerations for Information Planes. In *Workshop on Real, Large, Distributed Systems (WORLDS)*, Dec. 2004.

[20] D. D. Clark. The structuring of systems using upcalls. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 171–180, Dec. 1985.

[21] D. D. Clark. The design philosophy of the DARPA Internet protocols. In *ACM SIGCOMM*, pages 106–114, Aug. 1988.

[22] D. D. Clark, C. Partridge, J. C. Ramming, and J. T. Wroclawski. A knowledge plane for the Internet. In *ACM SIGCOMM*, pages 3–10, Aug. 2003.

[23] E. Cooke, R. Mortier, A. Donnelly, P. Barham, and R. Isaacs. Reclaiming network-wide visibility using ubiquitous end system monitors. In *USENIX Annual Technical Conference*, June 2006.

[24] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 161–174, Apr. 2008.

[25] DD-WRT firmware. `http://www.dd-wrt.com`.

[26] A. Dhamdhere, R. Teixeira, C. Dovrolis, and C. Diot. Troubleshooting network unreachabilities using end-to-end probes and routing data. In *ACM Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, Dec. 2007.

[27] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Transactions on Computers*, 52(2):99–112, Feb. 2003.

[28] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *ACM SIGCOMM*, pages 350–361, Aug. 2011.

[29] S. Goldberg, D. Xiao, E. Tromer, B. Barak, and J. Rexford. Path-quality monitoring in the presence of adversaries. In *SIGMETRICS*, pages 193–204, June 2008.

[30] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 202–210, Dec. 1989.

[31] 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 *ACM SIGCOMM*, pages 51–62, Aug. 2009.

[32] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary Internet end hosts. In *SIGCOMM Workshop on Internet Measurement (IMW)*, pages 5–18, Nov. 2002.

[33] G. Hamerly and C. Elkan. Bayesian approaches to failure prediction for disk drives. In *International Conference on Machine Learning (ICML)*, pages 202–209, June 2001.

[34] N. Hayashibara, X. Défago, R. Yared, and T. Katayama. The ϕ accrual failure detector. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 66–78, Oct. 2004.

[35] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX Annual Technical Conference*, pages 145–158, June 2010.

[36] T. Karagiannis, R. Mortier, and A. Rowstron. Network exception handlers: Host-network control in enterprise networks. In *ACM SIGCOMM*, Aug. 2008.

[37] D. Katz, K. Kompella, and D. Yeung. Traffic enginnering (TE) extensions to OSPF Version 2. RFC 3630, Network Working Group, Sept. 2003.

[38] D. Katz and D. Ward. Bidirectional forwarding detection (BFD) for IPv4 and IPv6 (single hop). RFC 5881, Network Working Group, June 2010.

[39] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: a scalable Ethernet architecture for large enterprises. In *ACM SIGCOMM*, pages 3–14, Aug. 2008.

[40] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. IP fault localization via risk modeling. In *Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[41] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Detection and localization of network black holes. In *INFOCOM*, pages 2180–2188, May 2007.

[42] R. Krishnan, J. P. G. Sterbenz, W. M. Eddy, C. Partridge, and M. Allman. Explicit transport error notification (ETEN) for error-prone wireless and satellite networks. *Computer Networks*, 46(3):343–362, 2004.

[43] A. Lakshman and P. Malik. Cassandra – A decentralized structured storage system. In *International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, Oct. 2009.

[44] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson,

S. Shenker, and I. Stoica. Achieving convergence-free routing using failure-carrying packets. In *ACM SIGCOMM*, pages 241–252, Aug. 2007.

[45] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998.

[46] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the FALCON spy network. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 279–294, Oct. 2011.

[47] A. Li, X. Yang, and D. Wetherall. SafeGuard: Safe forwarding during route changes. In *ACM Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, pages 301–312, Dec. 2009.

[48] J. Liu, S. Shenker, M. Schapira, and B. Yang. Ensuring connectivity via data plane mechanisms. In *Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2013.

[49] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 367–380, Nov. 2006.

[50] H. V. Madhyastha, E. Katz-Bassett, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane Nano: path prediction for peer-to-peer applications. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 137–152, Apr. 2009.

[51] J. Moy. OSPF version 2. RFC 2328, Network Working Group, Apr. 1998.

[52] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–41, Oct. 2011.

[53] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 17–28, Feb. 2007.

[54] J. Postel. Internet control message protocol. RFC 792, Network Working Group, 1981.

[55] The Quagga routing software suite. `http://www.nongnu.org/quagga/`.

[56] K. K. Ramakrishnan, S. Floyd, and D. Black. The addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, Network Working Group, Sept. 2001.

[57] A. Retana, L. Nguyen, R. White, A. Zinin, and D. McPherson. OSPF stub router advertisement. RFC 3137, Network Working Group, June 2001.

[58] T. Roscoe, R. Mortier, P. Jardetzky, and S. Hand. InfoSpect: Using a logic language for system health monitoring in distributed systems. In *ACM SIGOPS European Workshop*, pages 31–37, Sept. 2002.

[59] A. Shaikh, M. Goyal, A. Greenberg, R. Rajan, and K. K. Ramakrishnan. An OSPF topology server: design and evaluation. *IEEE JSAC*, 20(4):746–755, May 2002.

[60] A. Shaikh and A. Greenberg. OSPF monitoring: Architecture, design, and deployment experience. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 57–70, Mar. 2004.

[61] A. Shieh, E. G. Sirer, and F. B. Schneider. NetQuery: A knowledge plane for reasoning about network properties. In *ACM SIGCOMM*, pages 278–289, Aug. 2011.

[62] K. So and E. G. Sirer. Latency and bandwidth-minimizing failure detectors. In *European Conference on Computer Systems (EuroSys)*, pages 89–99, Mar. 2007.

[63] C. E. Stevens. AT attachment 8 - ATA/ATAPI command set. Technical Report 1699, Technical Committee T13, Sept. 2008.

[64] J. Stone and C. Partridge. When the CRC and TCP checksum disagree. In *ACM SIGCOMM*, pages 309–319, Aug. 2000.

[65] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An information plane for networked systems. In *ACM Workshop on Hot Topics in Networks (HotNets)*, Nov. 2003.

[66] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: A lightweight network location service without virtual coordinates. In *ACM SIGCOMM*, Aug. 2005.

[67] M. Zhang, C. Zhang, V. Pai, L. Peterson, and R. Wang. PlanetSeer: Internet path failure monitoring and characterization in wide-area services. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 167–182, Dec. 2004.

[68] X. Zhang, A. Jain, and A. Perrig. Packet-dropping adversary identification for data plane security. In *ACM Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, Dec. 2008.

[69] Y. Zhao, Y. Chen, and D. Bindel. Towards unbiased end-to-end network diagnosis. In *ACM SIGCOMM*, pages 219–230, Sept. 2006.

BOSS: Building Operating System Services

Stephen Dawson-Haggerty, Andrew Krioukov, Jay Taneja,
Sagar Karandikar, Gabe Fierro, Nikita Kitaev, and David Culler

Computer Science Division, University of California, Berkeley

Abstract

Commercial buildings are attractive targets for introducing innovative cyber-physical control systems, because they are already highly instrumented distributed systems which consume large quantities of energy. However, they are not currently programmable in a meaningful sense because each building is constructed with vertically integrated, closed subsystems and without uniform abstractions to write applications against. We develop a set of operating system services called BOSS, which supports multiple portable, fault-tolerant applications on top of the distributed physical resources present in large commercial buildings. We evaluate our system based on lessons learned from deployments of many novel applications in our test building, a four-year-old, 140,000sf building with modern digital controls, as well as partial deployments at other sites.

1 Introduction

Researchers and futurists working on ubiquitous and pervasive computing have long argued that a future full of personalized interaction between people and their environment is near [49, 42]. But this future has been primarily held back by the lack of a path from concept demonstration to broad deployment: developers have prototyped hundreds of interesting sensors [11, 36, 21], bringing new information about the world into a digital form, and tied these sensors together with actuators to provide interesting new capabilities to users. But invariably, these are developed and deployed as standalone, vertical applications, making it hard to share infrastructure investment among a variety of applications.

What is needed is an operating system to knit together existing pieces of infrastructure, Internet data feeds, and human feedback into a cohesive, extendable, and programmable system; *i.e.*, provide convenient abstractions and controlled access to shared physical resources. Doing so is a significant challenge, since such a system must bring together legacy systems with their own quirks, provide a path forward for new, native devices, and provide improved and simplified interfaces at multiple levels of abstraction. Existing buildings are not "programmable" in a meaningful sense: there are no layers of abstraction between the program and the system; programs may only access sensors and actuators at the very lowest level. As a result, applications are not *portable*, and it is impossible to provide *protected access* to an application, due to semantic mismatches between the level of policy and the level of access. We propose a new architecture for building control systems which, in addition to operating the machinery, provides for robust, portable application development and support many simultaneously running applications on the common physical infrastructure of a building. While buildings provide a concrete context, many of the ideas could be applied to other complex, connected physical systems.

We develop a collection of services forming a distributed operating system that solves several key problems that prevented earlier systems from scaling across the building stock. First, as buildings and their contents are fundamentally complicated, distributed systems with complex interrelationships, we develop a flexible approximate query language allowing applications to specify the components they interact with in terms of their relationship to other components, rather than specific hardware devices. Second, coordinated distributed control over a federated set of resources raises questions about behavior in the presence of failure. To resolve this concern, we present a transactional system for updating the state of multiple physical devices and reasoning about what will happen during a failure. Finally, there has previously been a separation between analytics, which deal with historical data, and control systems, which deal with real-time data. We demonstrate how to treat these uniformly in this environment, and present a time series service which allows applications to make identical use of both historical and real-time data.

Commercial buildings are an excellent environment in

which to investigate such new systems. Many buildings already contain thousands of sense and actuation points which can be manipulated to provide new and surprising experiences without hardware retrofits. They have large energy bills, consuming about 73% of all electricity in the United States [48], making energy efficiency a compelling incentive for new investment. Furthermore, they are large enough and contain enough people to drive issues of scale, partial failure, isolation, and privacy.

2 Existing Building Systems

A large modern commercial building represents the work of thousands of individuals and tens or hundreds of millions of dollars of investment. Most of these buildings contain extensive internal systems to manufacture an indoor environment: to provide thermal comfort (heating and cooling), good air quality (ventilation), and sufficient lighting; other systems provide for life safety (fire alarms, security) and connectivity (networking). These systems are frequently provided by different vendors and have little interoperability or extensibility beyond the scope of the original system design.

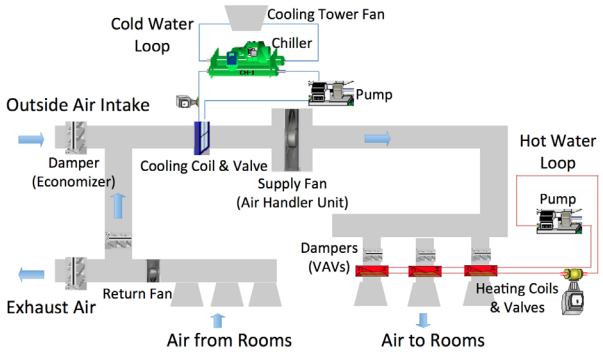


Figure 1: A typical process diagram of an HVAC system loop in a commercial building.

As an example of the complexity of many of these subsystems, Figure 1 shows one common design of a heating, ventilation, and air conditioning (HVAC) system for a large building. Air is blown through ducts, where it passes through variable air volume (VAV) boxes into internal rooms and other spaces. After circulating, it returns through a return air plenum where a portion is exhausted and the remaining portion is recirculated. The recirculated air is also mixed with fresh outside air, before being heated or cooled to a target supply temperature within an air handler unit (AHU), completing the loop. Other systems circulate hot and cold water for changing the air temperature. Many different control loops are present; the predominant control type is PID controllers used to meet setpoint targets for air pressure, temperature, and air volume.

This control in existing building systems operates on two levels. Direct control is performed in open and closed control loops between sensors and actuators: a piece of logic examines a set of input values and computes a control decision which commands an actuator. These direct control loops frequently have configuration parameters that govern their operation that are called setpoints; they are set by the building operator, installer, or engineer. Adjusting setpoints and schedules forms an outer logical loop, known as supervisory control. This logical distinction between types of control is typically reflected physically in the components and networking elements making up the system: direct control is performed by embedded devices called Programmable Logic Controllers (PLCs) that are hard-wired to sensors and actuators, while supervisory control and management of data for historical use is performed over a shared bus between the PLCs. This architecture is natural for implementing direct control loops since it minimizes the number of pieces of equipment and network links information must traverse to affect a particular control policy, making the system more robust, but it provides no coordinated control of distinct elements, and hard boundaries which are difficult to overcome. This collection of equipment is collectively known as a Building Management System (BMS).

The BMS is typically configured, managed, and programmed through a head-end node sitting on the shared bus. This node is responsible for providing an operator interface, storing historical "trend" data, and providing a point at which to reprogram the other controllers on the bus. It may also be a point of integration with other systems and therefore support some amount of reprogrammability or external access; for this reason, it can be a natural point of access to existing systems.

3 Design Development

To articulate the design of an operating system for buildings, we introduce three concrete, novel applications developed through research programs on our testbed building. The essential commonality is that all involve substantial interaction between components of building infrastructure, substantial computational elements, and building occupants, rather than simply providing a new interface to existing controls.

3.1 Motivating Applications

Ordinarily, the temperature within an HVAC zone is controlled to within a small range using a PID controller. The drive to reach an exact setpoint is actually quite inefficient, because it means that nearly every zone is heating or cooling at all times. A more relaxed strategy is one of *floating*: not attempting to effect the temperature of the room within a much wider band; however this is not one of the control policies available in typical commer-

cial systems even though numerous studies indicate that occupants can tolerate far more than the typical $2^{\circ}F$ variation allowed [4]. Furthermore, the minimum amount of ventilation air provided to each zone is also configured statically as a function of expected occupancy; however the actual requirement is stated in terms of fresh, outside air per occupant. The *HVAC optimization* application uses occupancy information derived from network activity, combined with information about the mix of fresh and return air currently in use to dynamically adjust the volume of ventilation air to each zone.

A second application was developed to improve comfort by giving occupants direct control of their spaces, inspired by previous work [16]. Using a smart-phone interface, the *personalized control* application gives occupants direct control of the lighting and HVAC systems in their workspaces. The application requires the ability to command the lights and thermostats in the space.

A third application is an *energy audit* application for our highly-instrumented building. Researchers input information about the structure of the building and the relationship between sensors and devices. This requires access to a uniform naming structure and streaming sensor data coming from physically-placed sensors. The former captures relationships between locations within the building, sensors, and loads (energy consumers) and the latter provides up-to-date information about physical measurements taken in locations throughout the building. This combination of data and metadata allow dashboarding and occupant feedback to access fine-grained slices of data – for instance, displaying the total energy consumed by all plug-loads on a particular floor.

Many other building applications have been developed in prior work including demand response [38], peak-price minimization [34], and occupant feedback [37]. All of these would benefit from a robust, common framework for controlling the building and the ability to run alongside other applications.

3.2 Architectural Implications

Experience with the *ad hoc* development of these kinds of applications led us to conclude that better abstractions and shared services would admit faster, easier, and richer application development, as well as a more fault tolerant system. The *HVAC optimization* application highlights the need for real-time access, locating the appropriate actuator for each temperature control unit, and replacing the local control logic with something new. Access to historical data is vital for training models and evaluating control strategies.

The *personalized climate control* application highlights the need for the ability to outsource control, at least temporarily, to a mobile web interface in a way that reverts gracefully to local control. It also integrates control over multiple subsystems that are frequently physically and logically separate in a building: HVAC and lighting.

The *energy audit* application highlights the need to couple semantic information with streaming sensor data in a uniform fashion and a way to meaningfully combine it with raw sensor data. It also emphasizes the need for real-time data cleaning and aggregation. Sensor feeds can be quite dirty, often missing values or containing errant data.

4 Design

The BOSS architecture consists of six main subsystems shown in Figure 2: (1) hardware abstraction and access abstraction; (2) naming and semantic modeling; (3) real-time time series processing and archiving; (4) a control transaction system; (5) authorization; and finally (6) running applications. The hardware abstraction layer elevates the plethora of underlying sensors and actuators to a shared, RESTful level and places all data within a shared global namespace, while the semantic modeling system allows for the description of relationships between the underlying sensors, actuators, and equipment. The time series processing system provides both real-time access to all underlying sensor data as well as stored historical data, and common analytical operators for cleaning and processing the data. The control transaction layer defines a robust interface for external processes wishing to control the system that is tolerant of failure and applies security policies. Last, "user processes" comprise the application layer. We expand on the design of each of these services below.

The mapping of these components to fault domains determines key properties of the overall system. The HPL must be physically co-located with the machinery it monitors and controls; failure of these components will prevent undoing actions taken by applications, although built-in control strategies provide the most basic level of fallback control. The transaction manager coordinates control over a set of HPL points and so it, combined with the set of HPL services it manages, determines a second, wider fault domain: this component forms the boundary at which other OS components can fail and still provide guaranteed behavior. The placement of the other services is flexible but impacts the availability of the resulting service; any other service failing could cause an application to crash.

4.1 Hardware Presentation Layer

Building systems are made up of a huge number of specialized sensors, actuators, communications links, and controller architectures. A significant challenge is overcoming this heterogeneity by providing uniform access to these resources and mapping them into corresponding virtual representations of underlying physical

Architectural component	Functional requirements	Placement
Hardware presentation layer	Expose the primitive low-level operations of hardware using a common interface.	Distributed as close to the physical sensors as possible (ideally, co-located).
Control transaction manager	Provide "all or nothing" semantics when applying control inputs; provide rollback of actions on failure, cancellation, or expiration.	Within the same failure domain as the HPL used to affect the changes.
Hardware abstraction layer	Map the low-level functions of the physical hardware to higher-level abstractions.	Anywhere.
Time series service	Maintain a history of readings from the sensors and actuators; provide application interface to data.	Replicated; may be offsite.
Authorization service	Approve application requests for access to building resources.	Anywhere.
Control processes	"User processes" implementing custom control logic.	Anywhere.

Table 1: Architectural components of a Building Operating System

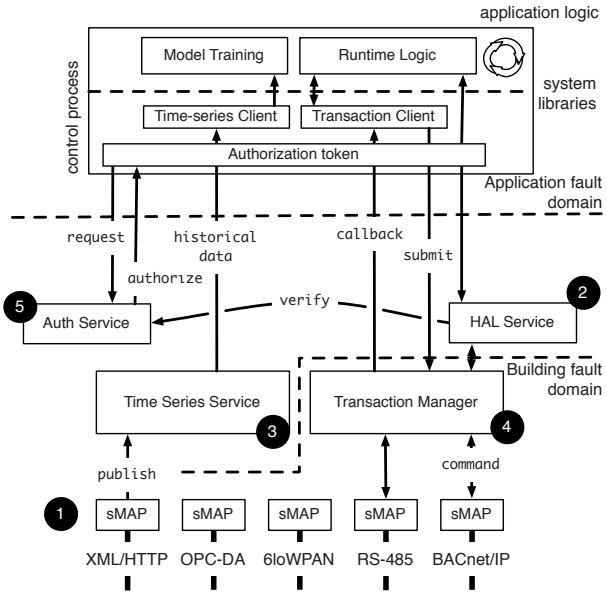


Figure 2: A schematic of important pieces in the system. BOSS consists of (1) the hardware presentation layer, the (2) hardware abstraction layer, the (3) time series service, and the (4) control transaction component. Finally, the (5) authorization service determines access controls. Control processes sit on top and consume these services.

hardware. At the lowest level is a Hardware Presentation Layer. The HPL hides the complexity and diversity of the underlying communications protocols and device interfaces, presenting hardware capabilities through a uniform, self-describing protocol. The HPL abstracts all sensing and actuation by mapping each individual sensor or actuator into a *point*: for instance, the temperature readings from a thermostat would be one sense point, while the damper position in a duct would be represented by an actuation point. These points produce *time series*, or streams, consisting of a timestamped sequence of readings of the current value of that point. The HPL provides a small set of common services for each sense and actuation point: the ability to read and write the point; the ability to subscribe to changes or receive periodic notifications about the point's value, and the ability to retrieve and append simple key-value structured metadata describing the point.

In order to provide the right building blocks for higher level functionality, this layer includes:

Naming: each sense or actuation point is named with a single, globally unique identifier. This provides canonical names for all data generated by that point for higher layers to use.

Metadata: most traditional protocols have limited or no metadata included about themselves, or their installation; however metadata, is incredibly important for the interpretation of data and for developing portable applications. The HPL allows us to include key-value metadata tags describing the data being collected to consumers.

Buffering and Leasing: many sources of data have the capability to buffer data for a period of time in case of the failure of the consumer; the HPL uses this to guard against missing data wherever possible. For actuators, safely commanding them in a fault tolerant way requires associating each write with a lease.

Discovery and Aggregation: sensors and the associated computing resources are often physically distributed with low-powered hardware. The HPL provides a mechanism to discover and aggregate many sensors into a single source on a platform with more resources, to support scalability.

This functionality is distributed across the computing resources closest to each sensor and actuator; ideally it is implemented natively by each device, although for legacy devices use a gateway or proxy. The HPL provides a small set of common services for each sense and actuation point: the ability to read and write the point; the ability to subscribe to changes or receive periodic notifications about the point's value, and the ability to retrieve and append simple key-value structured metadata describing the point.

4.2 Hardware Abstraction Layer

Unlike computer systems, buildings are nearly always custom-designed with unique architecture, siting, layout, mechanical and electrical systems, and control logic adapted to occupancy and local weather conditions. The

HAL allows applications to inspect these differences at a high level of abstraction, crucial for application portability. To do this, the HAL provides an approximate query language [28] allowing authors to describe the particular sensor or actuator that the application requires based on the relationship of that component to other items in the building, rather than hardcoding a name or tag. Applications can be written in terms of high-level queries such as "lights in room 410," rather than needing the exact network address of that point. The query language allows authors to search through multiple views of underlying building systems, including *spacial*, where objects are located in three-dimensional space; *electrical*, describing the electrical distribution tree; *HVAC*, describing how the mechanical systems interact; and *lighting*.

The HAL also abstracts the logic used to control building components such as pumps, fans, dampers, chillers, using a set of drivers to provide standard interfaces. Drivers provide high-level methods such as `set_speed` and `set_temperature` that are implemented using command sequences and control loops over the relevant HPL points. These drivers provide a place to implement device-specific logic that is needed to present standardized abstractions on top of eclectic hardware systems.

Drivers and applications use this functionality to determine locking sets, necessary for coexisting with other applications. For instance, an application that is characterizing the air handler behavior might want to ensure that the default control policy is in use, while varying a single input. It could use the approximate query language to lock all points on the air handler, excluding other processes. Since different models of the same piece of equipment may have different points even though they perform the same function, it is essential that applications can control sharing at the level of functional component rather than raw point name.

4.3 Time series service

Most sensors and embedded devices have neither the ability to store large quantities of historical data nor the processing resources to make use of them; such data are extremely important for historical analyses, model training, fault detection, and visualization. The challenge is storing large quantities of these data efficiently, while allowing applications to make the best use of them in near real-time for incorporation into control strategies. In existing systems, most historical data are goes unused because they are difficult to access and make sense of. Applications typically access data either by performing range queries over timestamps and streams, or by subscribing to the latest values. For instance, a typical query might train a model based room light-level readings for a period of one month, touching hundreds of millions of values. Even a modest-sized installation will easily have tens of billions of readings stored, with new data from the HPL mostly appended to the end of the time series. Finally, the data are usually dirty, often having desynchronized timestamps requiring outlier detection and recalibration before use.

The time series service (TSS) provides a low-latency application interface for accessing the large repository of stored data at different granularities. It consists of two parts: a stream selection language and a data transformation language. Using the stream selection language, applications can inspect and retrieve metadata about time series. The data transformation language allows clients to apply a pipeline of operators to the retrieved data to perform common data cleaning operations. This moves common yet complex processing logic out of applications, allowing them to focus on making the best use of the data, and also enables the possibility of optimizing common access patterns.

4.4 Control transactions

BOSS applications typically take the form of either coordinating control among multiple resources, which would otherwise operate independently as in the HVAC optimization, or extending control beyond the building to other systems, as in the personalized control or electric grid responsive control. The challenge is doing so in a way that is expressive enough to implement innovative new control algorithms, yet is robust to failure of network elements and controllers. Control algorithms that involve users or Internet-based data feeds should survive the failure of the parts of the control loop that run outside of the building without leaving any building equipment in an uncertain state. Therefore, we use a *transaction* metaphor for affecting changes to control state. Transactions in database systems are a way of reasoning about the consistency guarantees made when modifying multiple pieces of underling state; within BOSS, we use transactions as a way of reasoning about what happens when collections of control actions are performed.

A control transaction consists of a set of actions to be taken at a particular time: for instance, a coordinated write to multiple actuators. Actions at this level operate at the level of "points" – individual actuator outputs; the HAL uses its system model and driver logic to translate high-level requests into point-level operations that may include reads, writes, and locks.

To ensure reliability, control transactions require a *lease time* during which actions are valid, a *revert sequence* specifying how to undo the action, and an *error policy* stating what to do in case of a partial failure. When the lease expires, the transaction manager executes the revert sequence, which restores control of the system to the next scheduled direct controller. The ability to revert transactions provides the fundamental building

block for allowing control of the building to be turned over to more sophisticated and less-trusted applications, while providing baseline control. Actions may also be reverted on a partial failure, depending on the error policy; for instance, if the transaction manager cannot acquire a lock or the write to the underlying device fails. Revert sequences are provided for each action and can be thought of as the "inverse action" that undoes the control input. We require there to be a "lowest common denominator" control loop present that is able to run the building in its default (although potentially inefficient) operation. In this way, applications can always simply release control and the building will revert to its default control regime.

To support multiple applications, each point-level operation also is associated with a *priority level* and a *locking strategy*. These allow multiple higher-level processes or drivers to access the underlying points, while providing a mechanism for implicit coordination. Using a concept borrowed from BACnet, writes are performed into a "priority array" – a set of values that have been written to the point ordered by priority level. The actual output value is determined by taking the highest priority write. Although it provides for basic multiprocessing, the BACnet scheme has several problems. Without leases, a crashing application could leave the system locked in an uncertain state until its writes are manually cleared. Without notifications, it is difficult to determine if a particular write has been preempted by another process at a higher priority without periodically polling the array. The transaction manager adds notification, leasing and locking, allowing applications to be notified when their writes are preempted, or to prevent lower-priority processes from accessing the point.

4.5 Authorization service

In addition to providing high-level, expressive access to building systems, BOSS seeks to limit the ability of applications to manipulate physical resources. Most building operators will not turn over control to just anyone, and even for a trusted application developer, safeguards against runaway behavior are needed. The authorization service provides a means of authorizing principals to perform actions and is based on the approximate query language of the HAL; applications may be restricted by location (only lights on the fourth floor), value (cannot dim the lights below 50%), or schedule (access is only provided at night). This provides access control at the same semantic level as the operations to be performed.

BOSS checks access permissions on the level of individual method call and point name in the HAL and HPL using a two-stage approve/verify process. Applications first register their *intent* to access a point or method name with the service and what arguments they will call it with. The intents may either be automatically approved or presented to a building manager for approval. Security and safety checks are performed at time-of-use on each method call, providing the ability to revoke access. Verifying access permissions at time-of-use using an online server rather than at time-of-issue using signed capabilities has negative implications for availability and scalability, as it places the authorization service on the critical path of all application actions. However, we found the ability to provide definitive revocation a critical functionality necessary to convince building managers that the system is safe. This is one place where practical considerations of the domain won over our bias against adding more complexity to the command pathway.

4.6 Control processes

Updates to building control state are made atomically using control transactions; however, these are often part of larger, more complex long-lived blocks of logic. These is known as a "control process" (CP) and are analogous to a user process; each of our motiving applications is implemented as a control process in BOSS. CPs connect to services they require, such as the time series service, HAL, and transaction managers, and manage the input of control actions. Because of the careful design of transactions and the TSS, there are few constraints on where control processes can be placed in the computing infrastructure; if they fail or become partitioned from the actuator they control, the transaction manager will simply "roll back" their changes and revert to a lower-priority CP which has not experienced partition or failure and ultimately to the hard-coded control strategy.

5 Implementation and Evaluation

To evaluate our architecture for building software systems, we have developed a prototype implementation of the Building Operating System Services. BOSS implements all system components, and is currently being used by researchers in a living lab context. The system is built mostly in `Python`, with `C` used for certain performance-critical parts; all together, it is about 10,000 lines of non-application source code. The system uses a service-oriented design, with canonical DNS names used for finding other services; most services communicate using RESTful interfaces exchanging JSON objects. In cases where authentication is required, two-sided SSL is used.

We have fully installed BOSS on our test building, Sutardja Dai Hall: a four-year-old, 140,000 square foot building containing mostly open "collaboratory" spaces alongside faculty offices on the UC Berkeley campus. Additionally, we have partially installed BOSS in many other buildings; we have performed full BMS integration in two other campus buildings, and have used the

HPL for data collection and analysis in around 100 other buildings. The system is running many applications, including our motivating examples; we study these actual applications to illustrate how BOSS achieves its goals.

In our implementation, the HPL and the transaction manager run physically co-located with the BMS computer. Because the HPL buffers data when the time series service is unavailable and the transaction manager will revert any actions taken by failing or disconnect processes, this provides the best fault tolerance to failures outside of the building. Other services run mostly in a server room, although some applications are hosted on cloud services such as EC2. The failure or partition of an application from any of the services generally causes the application to fail, with the transaction manager reverting any actions taken once its leases expire.

5.1 Hardware Presentation

The presentation layer allows higher layers to retrieve data and command actuators in a uniform way. Our HPL is a revised version the Simple Measurement and Actuation Profile [12, 13], which provides RESTful access to data sources and actuators, exposing the command resource tree shown in Figure 3.

```
/data/                    # all timeseries and collections
    {
        "Contents" : ["sensor0"],
        "Metadata" : { "SourceName" : "Example sMAP Source" },
    }
/data/sensor0
    { "Contents" :["channel0"] },
/data/sensor0/channel0
    {
        "uuid" : "a7f63910-ddc6-11e0-8ab9-13c4da852bbc",
        "Readings" : [ [1315890624000, 12.5 ] ]
    }
/reports/                 # data destinations
```

Figure 3: Resource tree exported by sMAP. *Sensors and actuators are mapped to time series resources identified by UUIDs. Metadata from underlying systems are attached as key-value tags associated with time series or collections of time series.*

Ease of integration is key when interfacing with existing systems. The sMAP library[1] takes care of the mechanics of providing the external interface, and allows driver writers to focus on implementing only sensor or actuator-specific logic. It uses abstracted drivers that separate the device-specific logic needed for talking with a device (native communications protocols, *etc*) from the site-specific configuration (network locations, sampling rates, *etc*). To demonstrate its flexibility and versatility, we have implemented around 25 driver modules which integrate with three major BMS vendors, various low-power wireless devices, several different three-phase electric meters, weather stations, and detailed operations data from all major US electric grids to enable control strategies taking account of time-of-use pricing and renewable energy availability; together, more than 28,000 streams are present in the HPL[2]. We interface with the building management system of our test building primarily over BACnet.

5.2 Hardware Abstraction Layer

The hardware abstraction enables application portability in two ways: it supports queries over the relationships between building components, and provides drivers with standardized methods for making control inputs.

5.2.1 Semantic Query Language

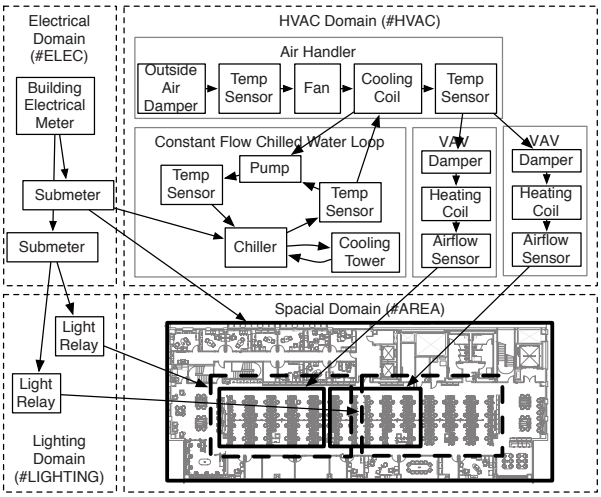


Figure 4: Partial functional and spacial representation of our test building. *Directed edges indicate a "supplies" or "feeds into" relationship. Queries are executed by searching the graph.*

The query interface allows applications to select objects based on type, attributes, and functional or spacial relationships, allowing programmers to describe the particular sensor or actuator that the application requires rather than hardcoding a name or tag. This allows applications to be portable across buildings with different designs[28].

Queries are expressed in terms of metadata tags from the HPL, and relationships indicated by $<$ and $>$ operators with $A > B$ meaning that A supplies or feeds into B. For example, an air handler might supply variable air volume (VAV) boxes that supply rooms; a whole building power meter may feed into multiple breaker panels that supply different floors. The execution engine evaluates queries by searching a directed graph of objects. Figure 4 shows a partial rendering of the functional and spacial relationship graphs. Objects are exposed by drivers and can be low-level (*e.g.*, damper, sensor, fan) or high-level (*e.g.*, air handler, chilled water loop). Directed edges in-

dicate the flow of physical media, *i.e.*air, water, electricity. Tags describe the object type and functionality. We also represent spatial areas, defined as polygons on floor maps of the building and stored in a GIS database.

5.2.2 Drivers

BOSS drivers are implemented as persistent objects within a global namespace; CPs obtain references to drivers using the semantic query language from the HAL. To make driver code reusable in an environment where very little can be assumed about the underlying technology, drivers present both uniform top-level interfaces to the components they represent, as well as a template-based bottom interface allowing them to be automatically instantiated when the underlying HAL points have been tagged with appropriate metadata.

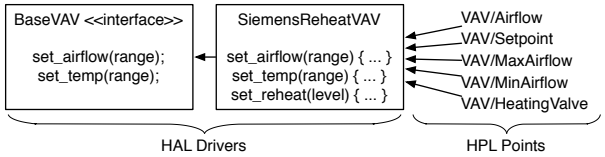


Figure 5: Drivers expose a set of methods on top, and are bound to HPL points based on metadata provided by the HPL. *Here, a Siemens VAV driver exposes the methods common to all VAVs as well as an additional* `set_reheat` *method.*

Figure 5 shows a schematic of a VAV driver: to illustrate these relationships, the `BaseVAV` interface which provides methods for controlling setpoint and temperature has been extended by the `SiemensReheatVAV` driver to provide an additional, non-standard `set_reheat` method. Drivers encapsulate device-specific logic needed to provide a standardized interface on top of the actual hardware present which may be more sophisticated than what is installed in a given building. For instance, the standard VAV control method allows applications to set a target temperature range (dead band). However, for many VAVs, including those in our test building, the width of this band is impossible to change once installed. Therefore, the driver emulates the desired behavior by manipulating the set point as the temperate moves in and out of the dead band.

Our driver system includes representations of many common building infrastructure elements such as VAVs, dampers, water pumps, air handlers, economizers, lights, light switches and transformers – and maps them into canonical representations that are placed into the metadata graph. Using drivers' bottom layer templates, appropriate drivers are loaded in places where HPL points have matching metadata, subject to hand checking.

5.3 Time Series Service

The time series service is responsible for **storing**, **selecting**, and **cleaning** both real-time and historical data. Figure 6 shows an example query exercising all three of these functions. The TSS contains three main components: the `readingdb` historian[3] provides compressed, low-latency, and high-throughput access to raw time series data. A selection engine performs SQL-to-SQL compilation on user queries, allowing them to flexibly select data streams on the basis of the tags applied by the HPL. A data transformation component applies domain-specific operations to the data.

5.3.1 readingdb

Many building energy products build on start SQL databases which are disappointing at scale. Figure 7 compares `readingdb` performance to MySQL (using both InnoDB and MyISAM storage engines) and PostgreSQL tuned for time series data on a representative usage pattern. Here, the database is loaded with synthetic data (simulating a trickle load), and periodically stopped to query a fixed-size set of data as well as to measure how large the stored data are on disk. Because MyISAM appends all records to the end of the volume, inserts are uniformly very cheap; however, query performance is poor and furthermore scales with the size of the database rather than the size of the results set. Postgres and InnoDB keep data ordered by time on disk resulting in more predictable query performance, but have more expensive insert paths; furthermore the B+-tree indexes scale poorly when presented with a large number of leaf keys. `readingdb`'s bucketing algorithm mitigates these issues (while still using an index for fast random access) by packing neighboring records together using only a single key, achieving an order of magnitude better compression than the other tree-based schemes.

5.3.2 Data Selection

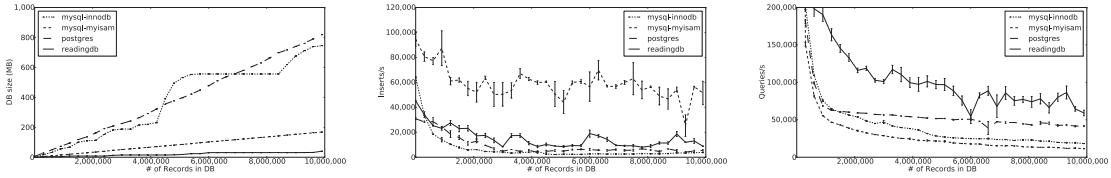
With tens of thousands of data streams present, finding the right one can be a challenge. The HPL provides the basis for this naming by identifying each data stream with a unique identifier (a `UUID`), and attaching key-value metadata to it. The time series service provides the mechanism to apply complex queries to these streams to locate them on the basis of the metadata, *i.e.*, an "entity-attribute-value" schema. Our system uses an SQL-to-SQL complier to transform logical queries in key-space to queries on the underlying database schema, allowing users to specify any attribute in the HPL. Line 3 in Figure 6 is an example of easily locating all datacenter power feeds.

5.3.3 Data Transformation

The processing pipeline allows operators to inspect both data (time, value vectors) as well as metadata: operators are first bound to the actual streams to be pro-

```
1: apply sum(axis=1) < missing < paste < window(mean, field="minute", width=15)
2: to data in ("4/20/2012", "4/21/2012")
3: where Metadata/Extra/System = 'datacenter' and Properties/UnitofMeasure = 'kW'
```

Figure 6: Example query executed by the time series service. *Line 1 uses a pipeline of four data cleaning operators to aggregate by resampling data in 15-minute intervals and then filtering missing data, Line 2 selects a range of time from the* `readingdb` *storage manager, and Line 3 queries metadata to locate datacenter power feeds.*



(a) Size of on-disk files (b) Insert performance (c) Query performance

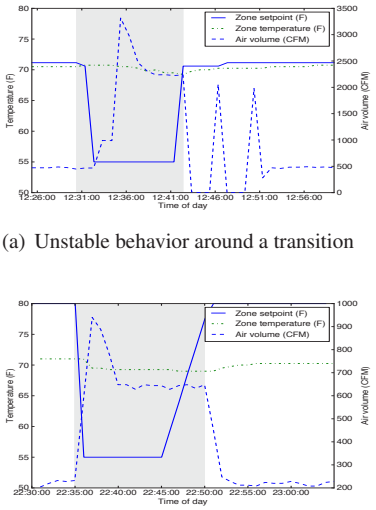
Figure 7: `readingdb` time series performance compared to two relational databases. *Compression keeps disk I/O to a minimum, while bucketing prevents updating the B+-tree indexes on the time dimension from becoming a bottleneck. Keeping data sorted by stream ID and timestamp preserves locality for range queries.*

cessed and since each operator can inspect the metadata of the input streams, it is possible to implement operators that transform data based on the metadata such as a unit or timezone conversion. Using simple combinations of these operators, queries can interpolate time-stamps, remove sections with missing data, and compute algebraic formulas over input data. Extending the set of operators is simple since we provide support for wrapping arbitrary `Python` functions which operate on vector data; in particular, we have imported most of the `numpy` numerical library automatically.

5.4 Transaction Manager

Submitted transactions become runnable once the start time has passed. The **scheduler** chooses the next action from among the runnable actions, taking into account considerations of both upper and lower layers. It considers the priorities of the various runnable actions as well as the concurrency requirements of the underlying hardware. For instance, some devices are present on a shared 9600-baud RS-485 bus; Internet-style applications can easily overwhelm such a limited resource. Priorities are implemented as separate FIFO queues at each priority level. Once actions are scheduled, actual execution is passed off to controller components that perform the action by communicating with the appropriate sMAP devices and proxies. When the lifetime of a transaction expires, it is canceled, or it encounters an error, the revert method is used to enqueue new commands to undo the previous control inputs.

The naïve reversion policy would simply clear any writes made; however, Figure 8(a) illustrates one problem with this method. Here, the setpoint is reduced at around 12:31, causing air volume to increase and room temperature to fall. However, when this change is reverted at 12:41, the default commercial controller which takes over becomes confused by the unexpected deviation from setpoint, causing the damper position (and thus air volume) to oscillate several times before finally stabilizing. Understanding and dealing with this issue is properly the concern of a higher-level component such as a VAV driver; to allow this, some drivers provide a custom revert action along with their inputs. These actions consist of restricted control sequences requiring no communication, replacing the default reversion policy. In Figure 8(b), the VAV driver uses a custom revert sequence to gradually release control.



(a) Unstable behavior around a transition

(b) Specialized reversion sequences can deal with this problem

Figure 8: Drivers may implement specialized reversion sequences to preserve system stability when changing control regimes.

6 Applications

We further evaluate BOSS in two ways: first, we examine how the system architecture makes implementing our three motivating applications simpler and more concise, while showing how it helps applications to coexist. Second, we provide a survey of other applications which have been implemented using BOSS, providing evidence of the system's generality.

6.1 HVAC Optimization

The HVAC optimization control process consists of two strategies: temperature floating and ventilation adjustment. Building codes often require a rate of fresh air ventilation per room based on occupancy and room size [9, 4]. Keeping ventilation rates at the required minimum is highly desirable for energy savings since it reduces fan power and the need for air conditioning; however, this is difficult to do in traditional building control systems because separate control loops are in charge of varying the fresh air intake into the building, controlling the per-room airflow, and detecting occupants. Occupancy detection is a well-researched subject that is best performed by fusing data from many sensors [2, 1, 31] not normally available.

Figure 9 shows pseudocode implementing the airflow reductions. The code uses the HAL semantic query interface to find all dampers controlling fresh air intake and adjusts the ventilation rates for their downstream rooms – the more fresh air being brought into the building from the outside, the less airflow is required per room to maintain the required freshness. In the example, line 3 returns dampers servicing the two air handlers (`AH1A` and `AH2A` in our building), each of which services around 70 zones, which are found on line 4. We use a simple occupancy model based on time of day and class schedule obtained from a Google Calendar feed, and scale the ventilation as a function of the number of people. This demonstrates coordinated control across traditionally independent building components: on line 6, the actual fresh air intake setting is used to control the room ventilation requirements. Furthermore, a separate building with completely different ventilation layout would be able to run virtually the same control application.

```
1 proc = BossProcess(timeout=15min, auth_token=ABC)
2 while True:
3   for dmp in hal.find('#OUT_AIR_DMP > #AH'):
4     for vav in hal.find('#VAV < $%s' % dmp.name):
5       occ = model.estimate_occupancy(vav)
6       vav.set_min_airflow((vav.min_fresh_air() /
7           dmp.get_percent_open()) * occ)
8   time.sleep(15*60)
```

Figure 9: Ventilation component of the HVAC optimization application.

6.2 Personalized Control

A second application, a personalized control system, takes direct occupant input to adjust room temperatures and ventilation. One of its key features is the ability to temporarily blast warm or cold air into the space in response to a user request. Fault tolerance is crucial in this application; blasts must be reverted even if the control process crashes to ensure occupant comfort and avoid wasting energy. Figure 10 shows the execution flow of the personalized control application and the error handling in response to an emulated crash.

The application writes to a room setpoint in response to a user request but shortly thereafter crashes. The transaction manager reverts the blast action by undoing the submitted transaction. A subplot of room temperature taken while executing this control flow is also shown in Figure 10. Temperature drops while the cold blast is running and reverts to normal after the application crashes. Unlike traditional computer systems, reverting the room temperature takes time as the space slowly warms back up to steady state.

We run the personalized control application concurrently with the HVAC optimization application. Since both apps access the same VAV point, some coordination is required to ensure correct behavior. In this case, the HVAC optimization application can coexist with the personal control application: if its commands are overridden at a higher priority, it simply regains control whenever the higher priority application is finished. However, the inverse situation is not acceptable: since users expect an immediate response when initiating a blast, the application locks the VAV points with an exclusive lock, so that if it is itself overridden, it will immediately cause the transaction to abort and display an error message.

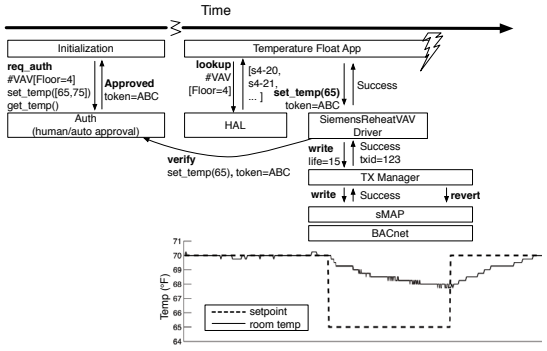


Figure 10: Execution flow of the personalized control application responding to a user request for cooling. *After the control process crashes, the transaction manager automatically reverts past actions.*

6.3 Auditing and Baselining

We use the auditing application to compute energy savings from the HVAC optimization and personal con-

trols. The HPL and time series service allow easy access to historical data from all meters within the building. We use these data to train a baseline model of power consumption of the building in its standard operating regime, regressing against outside air temperature, time of day, and class schedules. This model is used to produce a new real-time baseline stream which appears as a new virtual feed in the HPL. Using this baseline, we can compare building performance before and after enabling our optimization and control algorithms. Figure 11 shows measured power consumption and the modeled power baseline. Power consumption drops by 28kW, about 17%, after launching our optimization apps.

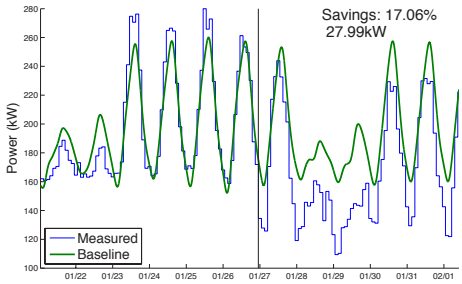


Figure 11: Energy use before and after starting our HVAC control applications in our test building showing energy savings of approximately 17%.

6.4 Deployment and Application Survey

To begin quantifying the generality of BOSS is, we surveyed a number of users inside and outside our group who have written applications to produce Table 2. These include the motivating applications above, as well as applications which perform model-predictive control of various system components and conduct comfort analyses of building data. Overall, application writers felt that their ability to spend time on their actual problems such as system modeling or producing visualizations of the data was much improved by operating at a higher level of abstraction; furthermore many appreciated the ability to write application code that might have bugs and yet be assured that the system would fail gracefully.

7 Related Work

7.1 Ubiquitous Computing

There have been many attempts to provide programmable abstractions to make it easier to run tasks on devices, predominantly in homes. For instance, ICrafter [39] integrates devices in an intelligent workspace into a service-oriented framework, used to generate user interfaces, while ubiHome [17] applies semantic web techniques to describe the services provided by a variety of consumer devices. Several "living laboratories" such as Sensor Andrew and HOBNET [41, 19] also work to make experimentation with ubiquitous computing environments simple and scalable, performing complimentary research on communications protocols.

Microsoft HomeOS [14] is a related attempt to provide high-level abstractions to make programming physical devices simpler. All intelligent devices in a home are presented as PC peripherals, and applications are implemented as modules loaded into a single system image; these applications are strongly isolated using run-time containers and `Datalog` descriptions of access rules. Our work takes a fundamentally different approach: we allow applications to be distributed and enforce safety at the transaction manager and driver services, at the cost of limiting control over the behavior of applications. By allowing control to be decentralized, we allow the system to be configured so as to trade off partition-tolerance with cost. Unlike HomeOS, we allow applications to be written in terms of components' relationship with other components, and provide efficient access to historical data; these functions are essential for scalability.

7.2 Metadata

Industry Foundation Classes [22] specify models for structural, mechanical, and electrical aspects of buildings. IFCs are intended to describe building design and facilitate sharing of information among design and construction teams. IFC includes classes for HVAC equipment and a connectivity model for building a directed graph of objects [8]. Project Haystack [40] uses a list of tags and rules about their use to describe building components. Liu, *et al.* [30] focus on integrating existing metadata from multiple sources and devising a common data representation for use by applications. Our work is complementary and focuses on how applications can conveniently make use of available building controls portably and at a higher level of abstraction.

7.3 Protocols

OLE for Process Control (OPC) is commonly used for controls interoperability. Based on DCOM, OPC accomplishes some of the same goals as the HPL [35], and contains a component for accessing historical data: OPC-HDA; however it does not provide the ability to apply analytical operators to the stored data and also can only locate data by point name. BACnet, or the Building Automation and Control Network protocol, also provides a standardized interface for accessing devices in a building [3]. It provides for the discovery of BACnet objects on a local subnet, and the ability for these objects to export standardized services, such as ReadValue and WriteValue. Other industrial controls protocols like WirelessHART [51], Modbus [33], and many others provide low-level access to field devices similar to the level of the HPL; however these form only the lowest-level building-blocks of a complete system. Protocols in the

Name	Description	Sensors Used	Actuators Used	Type of Control	HPL Adaptors	External Examples
Demand Ventilation	Ventilation rates are modulated to follow occupancy, and room temps can float at night.	VAV temps, airflow, & CO2	VAV Damper Positions	Supervisory	BACnet, 6loWPAN	
Supply Air Temp Control	Air Handling Unit supply air temp (AHU SAT) is optimized using model-predictive control (MPC) [5].	AHU SAT, VAV temps & airflow	AHU SAT	Supervisory	BACnet	[50], [29], [7], [52]
VAV Control	Individual variable air-volume boxes are exercised to create detailed models of their response.	VAV temps & airflow	VAV damper positions	Direct	BACnet	
Building Audit	Loads throughout the building are surveyed to enable DR and energy efficiency efforts.	Plug-load meters, submeters	N/A	N/A	ACme [25], BACnet	[18], [20], [32], [10], [45]
Personal Building Control	Users are presented with web-based lighting and HVAC control of building systems.	Light power, VAV temps	Light level, VAV airflow	Direct	BACnet	[27], [32], [23], [15]
Personal Comfort Toolkit	Enables analysis of air stratification in building system operation.	Zone temperatures, stratification	N/A	N/A	BACnet, CSV, 6loWPAN	
Demand Response	Responds to electric grid-initiated control signals requesting setback.	N/A	Airflow & temperature setpoints	Supervisory	BACnet	[26], [46]

Table 2: Deployed applications making use of BOSS.

residential space like uPNP [24] and Zigbee [44] tend to have higher-level interoperability as a goal, typically defining device profiles similar to our driver logic; this work is valuable for defining common device interfaces but does not address failures when these interfaces are implemented by coordinating multiple devices.

7.4 Building Controls

A number of Building Management Systems enable building engineers to create applications. The Siemens APOGEE [43] system provides the Powers Process Control Language (PPCL) for design of control applications; PPCL allows for custom logic, PID loops, and alarms. ALC's LogicBuilder [6] allows for graphical construction of control processes by sequencing "microblocks," or control functions, from a library; however, the library of control functions is not extensible. Further, both of these systems can only interact with equipment physically connected to the panel on which the code is installed, limiting the use of external information in control decisions. Tridium provides the Niagara AX framework [47] for designing Internet-connected applications using its HPL-like interfaces to building equipment and external data sources. However, Tridium provides no semantic information about its abstracted components, limiting application portability.

8 Conclusion

Our work is a re-imagining building control systems of the future: secure, modular, extensible, and networked. Many of the problems with SCADA systems which appear in the popular media with increasing frequency can be linked to system architecture designed for another world: one without ubiquitous connectivity where systems are designed and used in isolation from other systems. This view is simply not realistic, but fixing the problems requires a fundamental re-architecting of the control system, drawing on distributed systems and Internet design discipline to design a system which is robust even while subject to Internet-scale attacks. Although challenging, we do not see an alternative because it seems inevitable that control systems will become ever more networked to provide increased efficiency and flexibility; the only question is whether the result will be robust and conducive to application development.

BOSS begins to provide these properties by introducing flexibility in several new places where it did not previously exist: the HPL provides equal access to all of the underlying data, while the transaction and control process metaphors allow true applications to exist, and fail safely. Looking forward, many challenges remain. Inferring the HAL with minimum manual effort is an important step to enabling this architecture in existing buildings. Much better tools are needed for incorporating existing BIM models, and tools for checking the derived model against real sensor data will be crucial since drawings rarely reflect the true state of the world. The emergence of software-defined networks also presents an interesting avenue for future exploration: if control intent is expressed abstractly, SDNs might be used to enforce access control and quality-of-service guarantees.

Acknowledgements

Thanks to our shepherd, Richard Mortier, and the anonymous reviewers for helpful comments. Domenico Caramagno, Scott McNally, and Venzi Nikiforov's support was instrumental in allowing us gain access to real commercial building control systems. This project was supported in part by the National Science Foundation under grants CPS-0932209 (LoCal), CPS-0931843 (ActionWebs), and CPS-1239552 (SDB). The sMAP project is generously supported by the UC Berkeley Energy and Climate Research Innovation Seed Fund.

Notes

[1] http://code.google.com/p/smap-data
[2] http://www.openbms.org
[3] http://github.com/stevedh/readingdb

References

[1] AGARWAL, Y., BALAJI, B., DUTTA, S., GUPTA, R. K., AND WENG, T. Duty-cycling buildings aggressively: The next frontier in hvac control. In *IPSN/SPOTS 2011* (2011).

[2] AGARWAL, Y., BALAJI, B., GUPTA, R., LYLES, J., WEI, M., AND WENG, T. Occupancy-driven energy management for smart building automation. In *Proceedings of the 2nd ACM Workshop On Embedded Sensing Systems For Energy-Efficiency In Buildings* (2010).

[3] AMERICAN SOCIETY OF HEATING, REFRIGERATING AND AIR-CONDITIONING ENGINEERS. *ASHRAE Standard 135-1995: BACnet*. ASHRAE, Inc., 1995.

[4] AMERICAN SOCIETY OF HEATING, REFRIGERATING AND AIR-CONDITIONING ENGINEERS. *ASHRAE Standard 55-2010: Thermal Environmental Conditions for Human Occupancy*. ASHRAE, Inc., 2010.

[5] ASWANI, A., MASTER, N., TANEJA, J., KRIOUKOV, A., CULLER, D., AND TOMLIN, C. Energy-efficient building hvac control using hybrid system lbmpc. In *Proceedings of the IFAC Conference on Nonlinear Model Predictive Control* (2012).

[6] AUTOMATED LOGIC CORPORATION. Eikon LogicBuilder for WebCTRL. http://www.automatedlogic.com/product/eikon-logicbuilder-for-webctrl/.

[7] AVCI, M., ERKOC, M., RAHMANI, A., AND ASFOUR, S. Model predictive HVAC load control in buildings using real-time electricity pricing. *Energy and Buildings* (2013).

[8] BAZJANAC, V., FORESTER, J., HAVES, P., SUCIC, D., AND XU, P. HVAC component data modeling using industry foundation classes. In *System Simulation in Buildings* (2002).

[9] CA ENERGY COMMISSION. California's energy efficiency standards for residential and nonresidential buildings, 2008.

[10] CHENG, Y., CHEN, K., ZHANG, B., LIANG, C.-J. M., JIANG, X., AND ZHAO, F. Accurate Real-Time Occupant Energy-Footprinting in Commercial Buildings. In *Proc. of 4th ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings (BuildSys)* (2012).

[11] COHN, G., STUNTEBECK, E., PANDEY, J., OTIS, B., ABOWD, G. D., AND PATEL, S. N. SNUPI: sensor nodes utilizing powerline infrastructure. In *Proceedings of the 12th ACM international conference on Ubiquitous computing* (2010), pp. 159–168.

[12] DAWSON-HAGGERTY, S., JIANG, X., TOLLE, G., ORTIZ, J., AND CULLER, D. sMAP: a simple measurement and actuation profile for physical information. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems* (2010).

[13] DAWSON-HAGGERTY, S., KRIOUKOV, A., AND CULLER, D. E. Experiences integrating building data with smap. Tech. Rep. UCB/EECS-2012-21, EECS Department, University of California, Berkeley, Feb 2012.

[14] DIXON, C., MAHAJAN, R., AGARWAL, S., BRUSH, A. J., LEE, B., SAROIU, S., AND BAHL, P. An operating system for the home. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012).

[15] ERICKSON, V., AND CERPA, A. E. ThermoVote: Participatory Sensing for Efficient Building HVAC Conditioning. In *Proc. of 4th ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings (BuildSys)* (2012).

[16] FOUNTAIN, M., BRAGER, G., ARENS, E., BAUMAN, F., AND BENTON, C. Comport control for short-term occupancy. *Energy and Buildings 21*, 1 (1994), 1 – 13.

[17] HA, Y.-G., SOHN, J.-C., AND CHO, Y.-J. ubihome: An infrastructure for ubiquitous home network services. In *Proceedings of the IEEE International Symposium on Consumer Electronics* (2007).

[18] HAY, S., AND RICE, A. C. The Case for Apportionment. In *Proc. of 1st ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings (BuildSys)* (2009).

[19] Holistic platform design for smart buildings of the future internet.

[20] HSU, J., MOHAN, P., JIANG, X., ORTIZ, J., SHANKAR, S., DAWSON-HAGGERTY, S., AND CULLER, D. HBCI: Human-Building-Computer Interaction. In *Proc. of 2nd ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings (BuildSys)* (2010).

[21] HULL, B., BYCHKOVSKY, V., ZHANG, Y., CHEN, K., GORACZKO, M., MIU, A., SHIH, E., BALAKRISHNAN, H., AND MADDEN, S. Cartel: a distributed mobile sensor computing system. In *Proceedings of the 4th international conference on Embedded networked sensor systems* (2006).

[22] ISO. Industry Foundation Classes, Release 2x, 2005.

[23] JAZIZADEH, F., AND BECERIK-GERBER, B. Toward Adaptive Comfort Management in Office Buildings Using Participatory Sensing for End User Driven Control. In *Proc. of 4th ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings (BuildSys)* (2012).

[24] JERONIMO, M., AND WEAST, J. *UPnP Design by Example: A Software Developer's Guide to Universal Plug and Play*. Intel Press, 2003.

[25] JIANG, X., LY, M. V., TANEJA, J., DUTTA, P., AND CULLER, D. Experiences with a High-Fidelity Wireless Building Energy Auditing Network. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems* (2009).

[26] KRIOUKOV, A., ALSPAUGH, S., MOHAN, P., DAWSON-HAGGERTY, S., CULLER, D. E., AND KATZ, R. H. Design and Evaluation of an Energy Agile Computing Cluster. Tech. Rep. UCB/EECS-2012-13, EECS Department, University of California, Berkeley, 2012.

[27] KRIOUKOV, A., DAWSON-HAGGERTY, S., LEE, L., REHMANE, O., AND CULLER, D. A Living Laboratory Study in Personalized Automated Lighting Controls. In *Proc. of 3rd ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings (BuildSys)* (2011).

[28] KRIOUKOV, A., FIERRO, G., KITAEV, N., AND CULLER, D. Building application stack (BAS). In *Proceedings of the 4th ACM Workshop On Embedded Sensing Systems For Energy-Efficiency In Buildings* (2012).

[29] LIANG, C.-J. M., LIU, J., LUO, L., TERZIS, A., AND ZHAO, F. RACNet: A High-Fidelity Data Center Sensing Network. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys)* (2009).

[30] LIU, X., AKINCI, B., GARRETT, J. H., AND BERGES, M. Requirements for a formal approach to represent information exchange requirements of a self-managing framework for HVAC systems. In *ICCCBE* (2012).

[31] LU, J., SOOKOOR, T., SRINIVASAN, V., GAO, G., HOLBEN, B., STANKOVIC, J., FIELD, E., AND WHITEHOUSE, K. The smart thermostat: using occupancy sensors to save energy in homes. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems* (2010).

[32] MARCHIORI, A., HAN, Q., NAVIDI, W., AND EARLE, L. Building the Case For Automated Building Energy Management. In *Proc. of 4th ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings (BuildSys)* (2012).

[33] MODICON, INC., INDUSTRIAL AUTOMATION SYSTEMS. Modicon MODBUS Protocol Reference Guide, 1996.

[34] NGHIEM, T., BEHL, M., PAPPAS, G., AND MANGHARAM, R. Green scheduling: Scheduling of control systems for peak power reduction. In *Proceedings of the International Green Computing Conference and Workshops* (2011).

[35] OPC TASK FORCE. OPC common definitions and interfaces, 1998.

[36] PARK, T., LEE, J., HWANG, I., YOO, C., NACHMAN, L., AND SONG, J. E-gesture: a collaborative architecture for energy-efficient gesture recognition with hand-worn sensor and mobile devices. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems* (2011), ACM.

[37] PIETTE, M. A., KILICCOTE, S., AND GHATIKAR., G. Design of an energy and maintenance system user interface for building occupants. In *ASHRAE Transactions*, vol. 109, pp. 665–676.

[38] PIETTE, M. A., KILICCOTE, S., AND GHATIKAR., G. Design and implementation of an open, interoperable automated demand response infrastructure. In *Grid Interop Forum* (2007).

[39] PONNEKANTI, S. R., LEE, B., FOX, A., FOX, O., WINOGRAD, T., AND HANRAHAN, P. ICrafter : A service framework for ubiquitous computing environments. In *Proceedings of the 3rd International Conference on Ubiquitous Computing* (2001).

[40] Project Haystack. http://project-haystack.org/.

[41] ROWE, A., BERGES, M., BHATIA, G., GOLDMAN, E., RAJKUMAR, R., GARRETT, J. H., MOURA, J. M. F., AND SOIBELMAN, L. Sensor Andrew: Large-scale campus-wide sensing and actuation. *IBM Journal of Research and Development 55*, 1.2 (Jan. 2011), 6:1 – 6:14.

[42] SATYANARAYANAN, M. Pervasive computing: vision and challenges. *IEEE Personal Communications 8*, 4 (August 2001), 10 –17.

[43] SIEMENS. APOGEE Building Automation Software. `http://w3.usa.siemens.com/buildingtechnologies/us/en/building-automation-and-energy-management/apogee/pages/apogee.aspx`.

[44] *Smart Energy Profile 2.0*. Zigbee Alliance, 2010.

[45] TAHERIAN, S., PIAS, M., COULOURIS, G., AND CROWCROFT, J. Profiling Energy Use in Households and Office Spaces. In *Proceedings of the 3rd Int'l Conference on Future Energy Systems (ACM e-Energy)* (2012).

[46] TANEJA, J., CULLER, D., AND DUTTA, P. Towards Cooperative Grids: Sensor/Actuator Networks for Renewables Integration. In *Proceedings of the 1st IEEE Int'l Conference on Smart Grid Communications* (2010).

[47] TRIDIUM. NiagaraAX. `http://www.tridium.com/cs/products_/_services/niagaraax`.

[48] U.S. DEPARTMENT OF ENERGY. 2011 buildings energy data book, 2012.

[49] WEISER, M. The computer for the 21st century. *SIGMOBILE Mobile. Computer Communication Review 3*, 3 (July 1999), 3–11.

[50] WEN, Y.-J., DIBARTOLOMEO, D., AND RUBINSTEIN, F. Co-simulation Based Building Controls Implementation with Networked Sensors and Actuators. In *Proc. of 3rd ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings (BuildSys)* (2011).

[51] *WirelessHART*. HART Communication Foundation, 2009.

[52] LVAREZ, J., REDONDO, J., CAMPONOGARA, E., NORMEY-RICO, J., BERENGUEL, M., AND ORTIGOSA, P. Optimizing building comfort temperature regulation via model predictive control. *Energy and Buildings* (2013).

Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks

Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan
MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, Mass.
{keithw,anirudh,hari}@mit.edu

Abstract

Sprout is an end-to-end transport protocol for interactive applications that desire high throughput and low delay. Sprout works well over cellular wireless networks, where link speeds change dramatically with time, and current protocols build up multi-second queues in network gateways. Sprout does not use TCP-style reactive congestion control; instead the receiver observes the packet arrival times to infer the uncertain dynamics of the network path. This inference is used to forecast how many bytes may be sent by the sender, while bounding the risk that packets will be delayed inside the network for too long.

In evaluations on traces from four commercial LTE and 3G networks, Sprout, compared with Skype, reduced self-inflicted end-to-end delay by a factor of 7.9 and achieved 2.2× the transmitted bit rate on average. Compared with Google's Hangout, Sprout reduced delay by a factor of 7.2 while achieving 4.4× the bit rate, and compared with Apple's Facetime, Sprout reduced delay by a factor of 8.7 with 1.9× the bit rate.

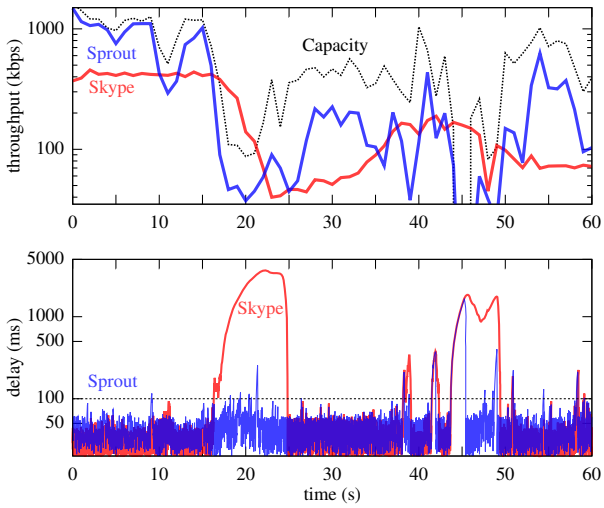
Although it is end-to-end, Sprout matched or outperformed TCP Cubic running over the CoDel active queue management algorithm, which requires changes to cellular carrier equipment to deploy. We also tested Sprout as a tunnel to carry competing interactive and bulk traffic (Skype and TCP Cubic), and found that Sprout was able to isolate client application flows from one another.

1 Introduction

Cellular wireless networks have become a dominant mode of Internet access. These mobile networks, which include LTE and 3G (UMTS and 1xEV-DO) services, present new challenges for network applications, because they behave differently from wireless LANs and from the Internet's traditional wired infrastructure.

Cellular wireless networks experience rapidly varying link rates and occasional multi-second outages in one or both directions, especially when the user is mobile. As a result, the time it takes to deliver a network-layer packet may vary significantly, and may include the effects of link-layer retransmissions. Moreover, these networks schedule transmissions after taking channel quality into account, and prefer to have packets waiting to be sent whenever a link is scheduled. They often achieve that goal by maintaining deep packet queues. The effect at the transport layer is that a stream of packets experiences widely varying packet delivery rates, as well as variable, sometimes multi-second, packet delays.

Figure 1: Skype and Sprout on the Verizon LTE downlink trace. For Skype, overshoots in throughput lead to large standing queues. Sprout tries to keep each packet's delay less than 100 ms with 95% probability.



For an interactive application such as a videoconferencing program that requires both high throughput and low delay, these conditions are challenging. If the application sends at too low a rate, it will waste the opportunity for higher-quality service when the link is doing well. But when the application sends too aggressively, it accumulates a queue of packets inside the network waiting to be transmitted across the cellular link, delaying subsequent packets. Such a queue can take several seconds to drain, destroying interactivity (see Figure 1).

Our experiments with Microsoft's Skype, Google's Hangout, and Apple's Facetime running over traces from commercial 3G and LTE networks show the shortcomings of the transport protocols in use and the lack of adaptation required for a good user experience. The transport protocols deal with rate variations in a reactive manner: they attempt to send at a particular rate, and if all goes well, they increase the rate and try again. They are slow to decrease their transmission rate when the link has deteriorated, and as a result they often create a large backlog of queued packets in the network. When that happens, only after several seconds and a user-visible outage do they switch to a lower rate.

This paper presents *Sprout*, a transport protocol designed for interactive applications on variable-quality

networks. Sprout uses the receiver's observed packet arrival times as the primary signal to determine how the network path is doing, rather than the packet loss, round-trip time, or one-way delay. Moreover, instead of the traditional reactive approach where the sender's window or rate increases or decreases in response to a congestion signal, the Sprout receiver makes a short-term forecast (at times in the near future) of the bottleneck link rate using probabilistic inference. From this model, the receiver predicts how many bytes are likely to cross the link within several intervals in the near future with at least 95% probability. The sender uses this forecast to transmit its data, bounding the risk that the queuing delay will exceed some threshold, and maximizing the achieved throughput within that constraint.

We conducted a trace-driven experimental evaluation (details in §5) using data collected from four different commercial cellular networks (Verizon's LTE and 3G 1xEV-DO, AT&T's LTE, and T-Mobile's 3G UMTS). We compared Sprout with Skype, Hangout, Facetime, and several TCP congestion-control algorithms, running in both directions (uplink and downlink).

The following table summarizes the average relative throughput improvement and reduction in self-inflicted queueing delay[1] for Sprout compared with the various other schemes, averaged over all four cellular networks in both directions. Metrics where Sprout did not outperform the existing algorithm are highlighted in red:

App/protocol	Avg. speedup vs. Sprout	Delay reduction (from avg. delay)
Sprout	1.0×	1.0× (0.32 s)
Skype	2.2×	7.9× (2.52 s)
Hangout	4.4×	7.2× (2.28 s)
Facetime	1.9×	8.7× (2.75 s)
Compound	1.3×	4.8× (1.53 s)
TCP Vegas	1.1×	2.1× (0.67 s)
LEDBAT	1.0×	2.8× (0.89 s)
Cubic	0.91×	79× (25 s)
Cubic-CoDel	0.70×	1.6× (0.50 s)

Cubic-CoDel indicates TCP Cubic running over the CoDel queue-management algorithm [17], which would be implemented in the carrier's cellular equipment to be deployed on a downlink, and in the baseband modem or radio-interface driver of a cellular phone for an uplink.

We also evaluated a simplified version of Sprout, called Sprout-EWMA, that tracks the network bitrate with a simple exponentially-weighted moving average, rather than making a cautious forecast of future packet deliveries with 95% probability.

Sprout and Sprout-EWMA represents different tradeoffs in their preference for throughput versus delay. As expected, Sprout-EWMA achieved greater throughput, but also greater delay, than Sprout. It outperformed TCP Cubic on both throughput and delay. Despite being end-to-end, Sprout-EWMA outperformed Cubic-over-CoDel on throughput and approached it on delay:

Protocol	Avg. speedup vs. Sprout-EWMA	Delay reduction (from avg. delay)
Sprout-EWMA	1.0×	1.0× (0.53 s)
Sprout	2.0×	0.60× (0.32 s)
Cubic	1.8×	48× (25 s)
Cubic-CoDel	1.3 ×	0.95× (0.50 s)

We also tested Sprout as a tunnel carrying competing traffic over a cellular network, with queue management performed at the tunnel endpoints based on the receiver's stochastic forecast about future link speeds. We found that Sprout could isolate interactive and bulk flows from one another, dramatically improving the performance of Skype when run at the same time as a TCP Cubic flow.

The source code for Sprout, our wireless network trace capture utility, and our trace-based network emulator is available at `http://alfalfa.mit.edu/`.

2 Context and Challenges

This section highlights the networking challenges in designing an adaptive transport protocol on cellular wireless networks. We discuss the queueing and scheduling mechanisms used in existing networks, present measurements of throughput and delay to illustrate the problems, and list the challenges.

2.1 Cellular Networks

At the link layer of a cellular wireless network, each device (user) experiences a different time-varying bit rate because of variations in the wireless channel; these variations are often exacerbated because of mobility. Bit rates are also usually different in the two directions of a link. One direction may experience an outage for a few seconds even when the other is functioning well. Variable link-layer bit rates cause the data rates at the transport layer to vary. In addition, as in other data networks, cross traffic caused by the arrival and departure of other users and their demands adds to the rate variability.

Most (in fact, all, to our knowledge) deployed cellular wireless networks enqueue each user's traffic in a separate queue. The base station schedules data transmissions taking both per-user (proportional) fairness and channel quality into consideration [3]. Typically, each user's device is scheduled for a fixed time slice over

[1] This metric expresses a lower bound on the amount of time necessary between a sender's input and receiver's output, so that the receiver can reconstruct more than 95% of the input signal. We define the metric more precisely in §5.

which a variable number of payload bits may be sent, depending on the channel conditions, and users are scheduled in roughly round-robin fashion. The isolation between users' queues means that the dominant factor in the end-to-end delay experienced by a user's packets is *self-interaction*, rather than cross traffic. If a user were to combine a high-throughput transfer and a delay-sensitive transfer, the commingling of these packets in the same queue would cause them to experience the same delay distributions. The impact of other users on delay is muted. However, competing demand can affect the throughput that a user receives.

Many cellular networks employ a non-trivial amount of packet buffering. For TCP congestion control with a small degree of statistical multiplexing, a good rule-of-thumb is that the buffering should not exceed the bandwidth-delay product of the connection. For cellular networks where the "bandwidth" may vary by two orders of magnitude within seconds, this guideline is not particularly useful. A "bufferbloated" [9] base station at one link rate may, within a short amount of time, be under-provisioned when the link rate suddenly increases, leading to extremely high IP-layer packet loss rates (this problem is observed in one provider [16]).

The high delays in cellular wireless networks cannot simply be blamed on bufferbloat, because there is no single buffer size that will always work. It is also not simply a question of using an appropriate Active Queue Management (AQM) scheme, because the difficulties in picking appropriate parameters are well-documented and become harder when the available rates change quickly, and such a scheme must be appropriate when applied to all applications, even if they desire bulk throughput. In §5, we evaluate CoDel [17], a recent AQM technique, together with a modern TCP variant (Cubic, which is the default in Linux), finding that on more than half of our tested network paths, CoDel slows down a bulk TCP transfer that has the link to itself.

By making changes—when possible—at endpoints instead of inside the network, diverse applications may have more freedom to choose their desired compromise between throughput and delay, compared with an AQM scheme that is applied uniformly to all flows.

Sprout is not a traditional congestion-control scheme, in that its focus is directed at adapting to varying link conditions, not to varying cross traffic that contends for the same bottleneck queue. Its improvements over existing schemes are found when queueing delays are imposed by one user's traffic. This is typically the case when the application is running on a mobile device, because cellular network operators generally maintain a separate queue for each customer, and the wireless link is typically the bottleneck. An important limitation of this approach is that in cases where these conditions don't hold, Sprout's traffic will experience the same delays as other flows.

2.2 Measurement Example

In our measurements, we recorded large swings in available throughput on mobile cellular links. Existing interactive transports do not handle these well. Figure 1 shows an illustrative section of our trace from the Verizon LTE downlink, whose capacity varied up and down by almost an order of magnitude within one second. From 15 to 25 seconds into the plot, and from 43 to 49 seconds, Skype overshoots the available link capacity, causing large standing queues that persist for several seconds, and leading to glitches or reduced interactivity for the users. By contrast, Sprout works to maximize the available throughput, while limiting the risk that any packet will wait in queue for more than 100 ms (dotted line). It also makes mistakes (e.g., it overshoots at $t = 43$ seconds), but then repairs them.

Network behavior like the above has motivated our development of Sprout and our efforts to deal explicitly with the uncertainty of future link speed variations.

2.3 Challenges

A good transport protocol for cellular wireless networks must overcome the following challenges:

1. It must cope with dramatic temporal variations in link rates.
2. It must avoid over-buffering and incurring high delays, but at the same time, if the rate were to increase, avoid under-utilization.
3. It must be able to handle outages without over-buffering, cope with asymmetric outages, and recover gracefully afterwards.

Our experimental results show that previous work (see §6) does not address these challenges satisfactorily. These methods are reactive, using packet losses, round-trip delays, and in some cases, one-way delays as the "signal" of how well the network is doing. In contrast, Sprout uses a different signal, the observed arrival times of packets at the receiver, over which it runs an inference procedure to make forecasts of future rates. We find that this method produces a good balance between throughput and delay under a wide range of conditions.

3 The Sprout Algorithm

Motivated by the varying capacity of cellular networks (as captured in Figure 1), we designed Sprout to compromise between two desires: achieving the highest possible throughput, while preventing packets from waiting too long in a network queue.

From the transport layer's perspective, a cellular network behaves differently from the Internet's traditional

infrastructure in several ways. One is that endpoints can no longer rely on packet drops to learn of unacceptable congestion along a network path ([9]), even after delays reach ten seconds or more. We designed Sprout not to depend on packet drops for information about the available throughput and the fullness of in-network queues.

Another distinguishing feature of cellular links is that users are rarely subject to standing queues accumulated by other users, because a cellular carrier generally provisions a separate uplink and downlink queue for each device in a cell. In a network where two independent users share access to a queue feeding into a bottleneck link, one user can inflict delays on another. No end-to-end protocol can provide low-delay service when a network queue is already full of somebody else's packets. But when queueing delays are largely self-inflicted, an end-to-end approach may be possible.[2]

In our measurements, we found that estimating the capacity (by which we mean the maximum possible bit rate or throughput) of cellular links is challenging, because they do not have a directly observable rate *per se*. Even in the middle of the night, when average throughput is high and an LTE device may be completely alone in its cell, packet arrivals on a saturated link do not follow an observable isochronicity. This is a roadblock for packet-pair techniques ([13]) and other schemes to measure the available throughput.

Figure 2 illustrates the *interarrival* distribution of 1.2 million MTU-sized packets received at a stationary cell phone whose downlink was saturated with these packets. For the vast majority of packet arrivals (the 99.99% that come within 20 ms of the previous packet), the distribution fits closely to a memoryless point process, or Poisson process, but with fat tails suggesting the impact of channel quality-dependent scheduling, the effect of other users, and channel outages, that yield interarrival times between 20 ms and as long as four seconds. Such a "switched" Poisson process produces a $1/f$ distribution, or *flicker noise*. The best fit is shown in the plot.[3]

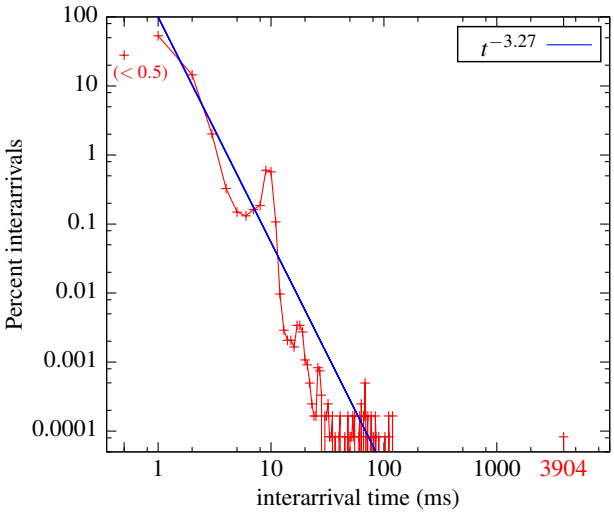
A Poisson process has an underlying rate λ, which may be estimated by counting the number of bits that arrive in a long interval and dividing by the duration of the interval. In practice, however, the rate of these cellular links varies more rapidly than the averaging interval necessary to achieve an acceptable estimate.

Sprout needs to be able to estimate the link speed, both now and in the future, in order to predict how many packets it is safe to send without risking their waiting in a network queue for too long. An uncertain estimate of future link speed is worth more caution than a certain estimate, so we need to quantify our uncertainty as well as our best guess.

[2] An end-to-end approach may also be feasible if all sources run the same protocol, but we do not investigate that hypothesis in this paper.

[3] We can't say exactly why the distribution should have this shape, but physical processes could produce such a distribution. Cell phones experience fading, or random variation of their channel quality with time, and cell towers attempt to send packets when a phone is at the apex of its channel quality compared with a longer-term average.

Figure 2: Interarrival times on a Verizon LTE downlink, with receiver stationary, fit to a $1/f$ noise distribution.



We therefore treat the problem in two parts. We model the link and estimate its behavior at any given time, preserving our full uncertainty. We then use the model to make forecasts about how many bytes the link will be willing to transmit from its queue in the near future. Most steps in this process can be precalculated at program startup, meaning that CPU usage (even at high throughputs) is less than 5% of a current Intel or AMD PC microprocessor. We have not tested Sprout on a CPU-constrained device or tried to optimize it fully.

3.1 Inferring the rate of a varying Poisson process

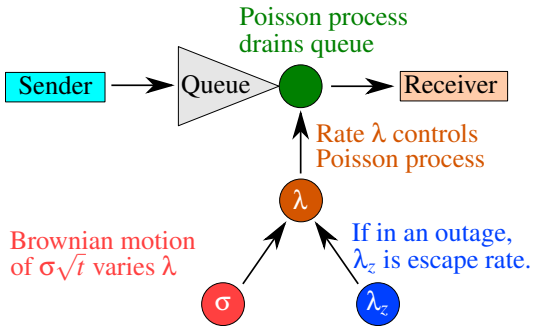
We model the link as a doubly-stochastic process, in which the underlying λ of the Poisson process itself varies in Brownian motion[4] with a noise power of σ (measured in units of packets per second per $\sqrt{\text{second}}$). In other words, if at time $t = 0$ the value of λ was known to be 137, then when $t = 1$ the probability distribution on λ is a normal distribution with mean 137 and standard deviation σ. The larger the value of σ, the more quickly our knowledge about λ becomes useless and the more cautious we have to be about inferring the link rate based on recent history.

Figure 3 illustrates this model. We refer to the Poisson process that dequeues and delivers packets as the service process, or packet-delivery process.

The model has one more behavior: if $\lambda = 0$ (an outage), it tends to stay in an outage. We expect the outage's

[4] This is a Cox model of packet arrivals [5, 18].

Figure 3: Sprout's model of the network path. A Sprout session maintains this model separately in each direction.



duration to follow an exponential distribution $\exp[-\lambda_z]$. We call λ_z the outage escape rate. This serves to match the behavior of real links, which do have "sticky" outages in our experience.

In our implementation of Sprout, σ and λ_z have fixed values that are the same for all runs and all networks. (σ = 200 MTU-sized packets per second per $\sqrt{\text{second}}$, and $\lambda_z = 1$.) These values were chosen based on preliminary empirical measurements, but the entire Sprout implementation including this model was frozen before we collected our measurement 3G and LTE traces and has not been tweaked to match them.

A more sophisticated system would allow σ and λ_z to vary slowly with time to better match more- or less-variable networks, Currently, the only parameter allowed to change with time, and the only one we need to infer in real time, is λ—the underlying, variable link rate.

To solve this inference problem tractably, Sprout discretizes the space of possible rates, λ, and assumes that:

- λ is one of 256 discrete values sampled uniformly from 0 to 1000 MTU-sized packets per second (11 Mbps; larger than the maximum rate we observed).
- At program startup, all values of λ are equally probable.
- An inference update procedure will run every 20 ms, known as a "tick". (We pick 20 ms for computational efficiency.)

By assuming an equal time between updates to the probability distribution, Sprout can precompute the normal distribution with standard deviation to match the Brownian motion per tick.

3.2 Evolution of the probability distribution on λ

Sprout maintains the probability distribution on λ in 256 floating-point values summing to unity. At every tick, Sprout does three things:

1. It *evolves* the probability distribution to the current time, by applying Brownian motion to each of the 255 values $\lambda \neq 0$. For $\lambda = 0$, we apply Brownian motion, but also use the outage escape rate to bias the evolution towards remaining at $\lambda = 0$.

2. It *observes* the number of bytes that actually came in during the most recent tick. This step multiplies each probability by the likelihood that a Poisson distribution with the corresponding rate would have produced the observed count during a tick. Suppose the duration of a tick is τ seconds (e.g., $\tau = 0.02$) and k bytes were observed during the tick. Then, Sprout updates the (non-normalized) estimate of the probabilities F:

$$F(x) \leftarrow \mathbb{P}_{\text{old}}(\lambda = x)\frac{(x\cdot\tau)^k}{k!}\exp[-x\cdot\tau].$$

3. It *normalizes* the 256 probabilities so that they sum to unity:

$$\mathbb{P}_{\text{new}}(\lambda = x) \leftarrow \frac{F(x)}{\sum_i F(i)}.$$

These steps constitute Bayesian updating of the probability distribution on the current value of λ.

One important practical difficulty concerns how to deal with the situation where the queue is underflowing because the sender has not sent enough. To the receiver, this case is indistinguishable from an outage of the service process, because in either case the receiver doesn't get any packets.

We use two techniques to solve this problem. First, in each outgoing packet, the sender marks its expected "time-to-next" outgoing packet. For a flight of several packets, the time-to-next will be zero for all but the last packet. When the receiver's most recently-received packet has a nonzero time-to-next, it skips the "observation" process described above until this timer expires. Thus, this "time-to-next" marking allows the receiver to avoid mistakenly observing that zero packets were deliverable during the most recent tick, when in truth the queue is simply empty.

Second, the sender sends regular heartbeat packets when idle to help the receiver learn that it is not in an outage. Even one tiny packet does much to dispel this ambiguity.

3.3 Making the packet delivery forecast

Given a probability distribution on λ, Sprout's receiver would like to predict how much data it will be safe for the sender to send without risking that packets will be stuck in the queue for too long. No forecast can be absolutely safe, but for typical interactive applications we would like to bound the risk of a packet's getting queued for longer than the sender's tolerance to be less than 5%.

To do this, Sprout calculates a *packet delivery forecast*: a cautious estimate, at the 5th percentile, of how many bytes will arrive at its receiver during the next eight ticks, or 160 ms.

It does this by *evolving* the probability distribution forward (without observation) to each of the eight ticks in the forecast. At each tick, Sprout sums over each λ to find the probability distribution of the cumulative number of packets that will have been drained by that point in time. We take the 5th percentile of this distribution as the cautious forecast for each tick. Most of these steps can also be precalculated, so the only work at runtime is to take a weighted sum over each λ.

3.4 The control protocol

The Sprout receiver sends a new forecast to the sender by piggybacking it onto its own outgoing packets.

In addition to the predicted packet deliveries, the forecast also contains a count of the total number of bytes the receiver has received so far in the connection or has written off as lost. This total helps the sender estimate how many bytes are in the queue (by subtracting it from its own count of bytes that have been sent).

In order to help the receiver calculate this number and detect losses quickly, the sender includes two fields in every outgoing packet: a sequence number that counts the number of bytes sent so far, and a "throwaway number" that specifies the sequence number offset of the *most recent* sent packet that was sent more than 10 ms prior.

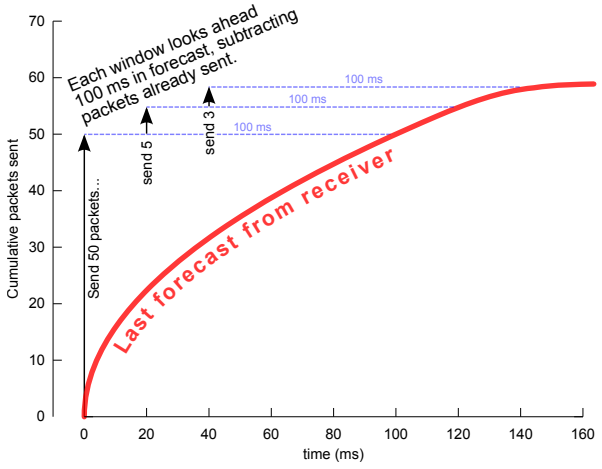
The assumption underlying this method is that while the network may reorder packets, it will not reorder two packets that were sent more than 10 ms apart. Thus, once the receiver actually gets a packet from the sender, it can mark all bytes (up to the sequence number of the first packet sent within 10 ms) as received or lost, and only keep track of more recent packets.

3.5 Using the forecast

The Sprout sender uses the most recent forecast it has obtained from the receiver to calculate a window size—the number of bytes it may safely transmit, while ensuring that every packet has 95% probability of clearing the queue within 100 ms (a conventional standard for interactivity). Upon receipt of the forecast, the sender timestamps it and estimates the current queue occupancy, based on the difference between the number of bytes it has sent so far and the "received-or-lost" sequence number in the forecast.

The sender maintains its estimate of queue occupancy going forward. For every byte it sends, it increments the estimate. Every time it advances into a new tick of the 8-tick forecast, it decrements the estimate by the amount of the forecast, bounding the estimate below at zero packets.

Figure 4: Calculating the window sizes from the forecast. The forecast represents the receiver's estimate of a lower bound (with 95% probability) on the cumulative number of packets that will be delivered over time.



To calculate a window size that is safe for the application to send, Sprout looks ahead five ticks (100 ms) into the forecast's future, and counts the number of bytes expected to be drained from the queue over that time. Then it subtracts the current queue occupancy estimate. Anything left over is "safe to send"—bytes that we expect to be cleared from the queue within 100 ms, even taking into account the queue's current contents. This evolving window size governs how much the application may transmit. Figure 4 illustrates this process schematically.

As time passes, the sender may look ahead further and further into the forecast (until it reaches 160 ms), even without receiving an update from the receiver. In this manner, Sprout combines elements of pacing with window-based flow control.

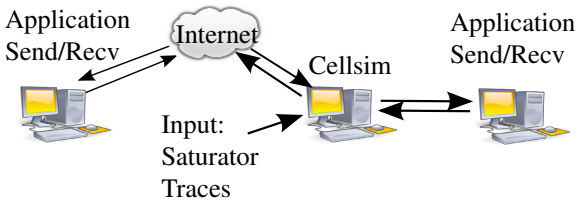
4 Experimental Testbed

We use trace-driven emulation to evaluate Sprout and compare it with other applications and protocols under reproducible network conditions. Our goal is to capture the variability of cellular networks in our experiments and to evaluate each scheme under the same set of time-varying conditions.

4.1 Saturator

Our strategy is to characterize the behavior of a cellular network by saturating its uplink and downlink at the same time with MTU-sized packets, so that neither queue goes empty. We record the times that packets actually cross the link, and we treat these as the ground truth representing all the times that packets *could* cross the link as long as a sender maintains a backlogged queue.

Figure 5: Block diagram of Cellsim



Because even TCP does not reliably keep highly variable links saturated, we developed our own tool. The Saturator runs on a laptop tethered to a cell phone (which can be used while in a car) and on a server that has a good, low-delay ($<$ 20 ms) Internet path to the cellular carrier.

The sender keeps a window of N packets in flight to the receiver, and adjusts N in order to keep the observed RTT greater than 750 ms (but less than 3000 ms). The theory of operation is that if packets are seeing more than 750 ms of queueing delay, the link is not starving for offered load. (We do not exceed 3000 ms of delay because the cellular network may start throttling or dropping packets.)

There is a challenge in running this system in two directions at once (uplink and downlink), because if both links are backlogged by multiple seconds, feedback arrives too slowly to reliably keep both links saturated. Thus, the Saturator laptop is actually connected to *two* cell phones. One acts as the "device under test," and its uplink and downlink are saturated. The second cell phone is used only for short feedback packets and is otherwise kept unloaded. In our experiments, the "feedback phone" was on Verizon's LTE network, which provided satisfactory performance: generally about 20 ms delay back to the server at MIT.

We collected data from four commercial cellular networks: Verizon Wireless's LTE and 3G (1xEV-DO / eHRPD) services, AT&T's LTE service, and T-Mobile's 3G (UMTS) service.[5] We drove around the greater Boston area at rush hour and in the evening while recording the timings of packet arrivals from each network, gathering about 17 minutes of data from each. Because the traces were collected at different times and places, the measurements cannot be used to compare different commercial services head-to-head.

For the device under test, the Verizon LTE and 1xEV-DO (3G) traces used a Samsung Galaxy Nexus smartphone running Android 4.0. The AT&T trace used a Samsung Galaxy S3 smartphone running Android 4.0. The T-Mobile trace used a Samsung Nexus S smartphone running Android 4.1.

[5]We also attempted a measurement on Sprint's 3G (1xEV-DO) service, but the results contained several lengthy outages and were not further analyzed.

Figure 6: Software versions tested

Program	Version	OS	Endpoints
Skype	5.10.0.116	Windows 7	Core i7 PC
Hangout	as of 9/2012	Windows 7	Core i7 PC
Facetime	2.0 (1070)	OS X 10.8.1	MB Pro (2.3 GHz i7), MB Air (1.8 GHz i5)
TCP Cubic	in Linux 3.2.0		Core i7 PC
TCP Vegas	in Linux 3.2.0		Core i7 PC
LEDBAT	in μTP	Linux 3.2.0	Core i7 PC
Compound TCP	in Windows 7		Core i7 PC

4.2 Cellsim

We then replay the traces in a network emulator, which we call Cellsim (Figure 5). It runs on a PC and takes in packets on two Ethernet interfaces, delays them for a configurable amount of time (the propagation delay), and adds them to the tail of a queue. Cellsim releases packets from the head of the queue to the other network interface according to the same trace that was previously recorded by Saturator. If a scheduled packet delivery occurs while the queue is empty, nothing happens and the opportunity to delivery a packet is wasted.[6]

Empirically, we measure a one-way delay of about 20 ms in each direction on our cellular links (by sending a single packet in one direction on the uplink or downlink back to a desktop with good Internet service). All our experiments are done with this propagation delay, or in other words a 40 ms minimum RTT.

Cellsim serves as a transparent Ethernet bridge for a Mac or PC under test. A second computer (which runs the other end of the connection) is connected directly to the Internet. Cellsim and the second computer receive their Internet service from the same gigabit Ethernet switch.

We tested the latest (September 2012) real-time implementations of all the applications and protocols (Skype, Facetime, etc.) running on separate late-model Macs or PCs (Figure 6).

We also added stochastic packet losses to Cellsim to study Sprout's loss resilience. Here, Cellsim drops packets from the tail of the queue according to a specified random drop rate. This approach emulates, in a coarse manner, cellular networks that do not have deep packet buffers (e.g., Clearwire, as reported in [16]). Cellsim also includes an optional implementation of CoDel, based on the pseudocode in [17].

4.3 SproutTunnel

We implemented a UDP tunnel that uses Sprout to carry arbitrary traffic (e.g. TCP, videoconferencing protocols) across a cellular link between a mobile user and a well-connected host, which acts as a relay for the user's Inter-

[6]This accounting is done on a per-byte basis. If the queue contains 15 100-byte packets, they will all be released when the trace records delivery of a single 1500-byte (MTU-sized) packet.

net traffic. SproutTunnel provides each flow with the abstraction of a low-delay connection, without modifying carrier equipment. It does this by separating each flow into its own queue, and filling up the Sprout window in round-robin fashion among the flows that have pending data.

The total queue length of all flows is limited to the receiver's most recent estimate of the number of packets that can be delivered over the life of the forecast. When the queue lengths exceed this value, the tunnel endpoints drop packets from the head of the longest queue. This algorithm serves as a dynamic traffic-shaping or active-queue-management scheme that adjusts the amount of buffering to the predicted channel conditions.

5 Evaluation

This section presents our experimental results obtained using the testbed described in §4. We start by motivating and defining the two main metrics: throughput and self-inflicted delay. We then compare Sprout with Skype, Facetime, and Hangout, focusing on how the different rate control algorithms used by these systems affect the metrics of interest. We compare against the delay-triggered congestion control algorithms TCP Vegas and LEDBAT, as well as the default TCP in Linux, Cubic, which does not use delay as a congestion signal, and Compound TCP, the default in some versions of Windows.

We also evaluate a simplified version of Sprout, called Sprout-EWMA, that eliminates the cautious packet-delivery forecast in favor of an exponentially-weighted moving average of observed throughput. We compare both versions of Sprout with a queue-management technique that must be deployed on network infrastructure. We also measure Sprout's performance in the presence of packet loss.

Finally, we evaluate the performance of competing flows (TCP Cubic and Skype) running over the Verizon LTE downlink, with and without SproutTunnel.

The implementation of Sprout (including the tuning parameters $\sigma = 200$ and $\lambda_z = 1$) was frozen before collecting the network traces, and has not been tweaked.

5.1 Metrics

We are interested in performance metrics appropriate for a real-time interactive application. In our evaluation, we report the *average throughput* achieved and the 95th-percentile *self-inflicted delay* incurred by each protocol, based on measurement at the Cellsim.

The *throughput* is the total number of bits received by an application, divided by the duration of the experiment. We use this as a measure of bulk transfer rate.

The *self-inflicted delay* is a lower bound on the end-to-end delay that must be experienced between a sender and receiver, given observed network behavior. We define it as follows: At any point in time, we find the most recently-sent packet to have arrived at the receiver. The amount of time since this packet was sent is a lower bound on the instantaneous delay that must exist between the sender's input and receiver's output in order to avoid a gap in playback or other glitch at this moment. We calculate this instantaneous delay for each moment in time. The 95th percentile of this function (taken over the entire trace) is the amount of delay that must exist between the input and output so that the receiver can recover 95% of the input signal by the time of playback. We refer to this as "95% end-to-end delay."

For a given trace, there is a lower limit on the 95% end-to-end delay that can be achieved even by an omniscient protocol: one that sends packets timed to arrive exactly when the network is ready to dequeue and transmit a packet. This "omniscient" protocol will achieve 100% of the available throughput of the link and its packets will never sit in a queue. Even so, the "omniscient" protocol will have fluctuations in its 95% end-to-end delay, because the link may have delivery outages. If the link does not deliver any packets for 5 seconds, there must be at least 5 seconds of end-to-end delay to avoid a glitch, no matter how smart the protocol is.[7]

The difference between the 95% end-to-end delay measured for a particular protocol and for an "omniscient" one is known as the *self-inflicted delay*. This is the appropriate figure to assess a real-time interactive protocol's ability to compromise between throughput and the delay experienced by users.

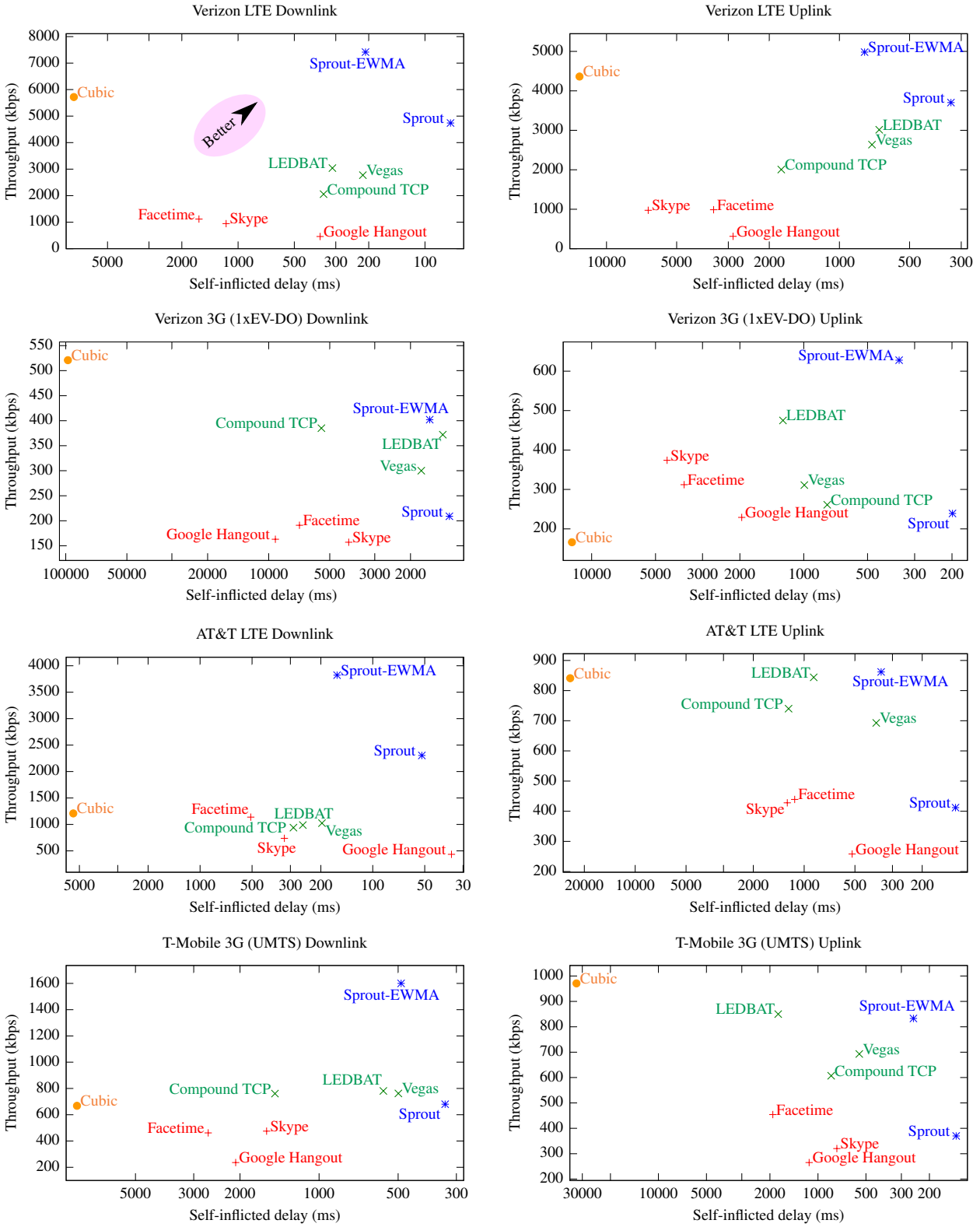
To reduce startup effects when measuring the average throughput and self-inflicted delay from an application, we skip the first minute of each application's run.

5.2 Comparative performance

Figure 7 presents the results of our trace-driven experiments for each transport protocol. The figure shows eight charts, one for each of the four measured networks, and for each data transfer direction (downlink and uplink). On each chart, we plot one point per application or protocol, corresponding to its measured throughput and self-inflicted delay combination. For interactive applications, high throughput and low delay (up and to the right) are the desirable properties. The table in the introduction shows the average of these results, taken over all the measured networks and directions, in terms of the average relative throughput gain and delay reduction achieved by Sprout.

[7] If packets are not reordered by the network, the definition becomes simpler. At each instant that a packet arrives, the end-to-end delay is equal to the delay experienced by that packet. Starting from this value, the end-to-end delay increases linearly at a rate of 1 s/s, until the next packet arrives. The 95th percentile of this function is the 95% end-to-end delay.

Figure 7: Throughput and delay of each protocol over the traced cellular links. Better results are up and to the right.



We found that Sprout had the lowest, or close to the lowest, delay across each of the eight links. On average delay, Sprout was lower than every other protocol. On average throughput, Sprout outperformed every other protocol except for Sprout-EWMA and TCP Cubic.

We also observe that Skype, Facetime, and Google Hangout all have lower throughput and higher delay than the TCP congestion-control algorithms. We believe this is because they do not react to rate increases and decreases quickly enough, perhaps because they are unable to change the encoding rapidly, or unwilling for perceptual reasons.[8] By continuing to send when the network has dramatically slowed, these programs induce high delays that destroy interactivity.

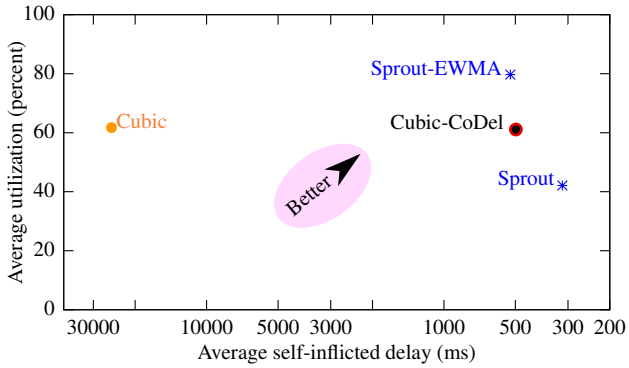
5.3 Benefits of forecasting

Sprout differs from the other approaches in two significant ways: first, it uses the packet arrival process at the receiver as the "signal" for its control algorithm (as opposed to one-way delays as in LEDBAT or packet losses or round-trip delays in other protocols), and second, it models the arrivals as a flicker-noise process to perform Bayesian inference on the underlying rate. A natural question that arises is what the benefits of Sprout's forecasting are. To answer this question, we developed a simple variant of Sprout, which we call *Sprout-EWMA*. Sprout-EWMA uses the packet arrival times, but rather than do any inference with that data, simply passes them through an exponentially-weighted moving average (EWMA) to produce an evolving smoothed rate estimate. Instead of a cautious "95%-certain" forecast, Sprout-EWMA simply predicts that the link will continue at that speed for the next eight ticks. The rest of the protocol is the same as Sprout.

The Sprout-EWMA results in the eight charts in Figure 7 show how this protocol performs. First, it outperforms all the methods in throughput, including recent TCPs such as Compound TCP and Cubic. These results also highlight the role of cautious forecasting: the self-inflicted delay is significantly lower for Sprout compared with Sprout-EWMA. TCP Vegas also achieves lower delay on average than Sprout-EWMA. The reason is that an EWMA is a low-pass filter, which does not immediately respond to sudden rate reductions or outages (the tails seen in Figure 2). Though these occur with low probability, when they do occur, queues build up and take a significant amount of time to dissipate. Sprout's forecasts provide a conservative trade-off between throughput and delay: keeping delays low, but missing legitimate opportunities to send packets, preferring to avoid the risk of filling up queues. Because the resulting throughput is relatively high, we believe it is a good choice for interactive applications. An application that is interested only in high throughput with less of an emphasis on low delay may prefer Sprout-EWMA.

[8] We found that the complexity of the video signal did not seem to affect these programs' transmitted throughputs. On fast network paths, Skype uses up to 5 Mbps even when the image is static.

Figure 8: Average utilization and delay of each scheme. Utilization is the average fraction of the cellular link's maximum capacity that the scheme achieved.



5.4 Comparison with in-network changes

We compared Sprout's end-to-end inference approach against an in-network deployment of active queue management. We added the CoDel AQM algorithm [17] to Cellsim's uplink and downlink queue, to simulate what would happen if a cellular carrier installed this algorithm inside its base stations and in the baseband modems or radio-interface drivers on a cellular phone.

The average results are shown in Figure 8. Averaged across the eight cellular links, CoDel dramatically reduces the delay incurred by Cubic, at little cost to throughput.

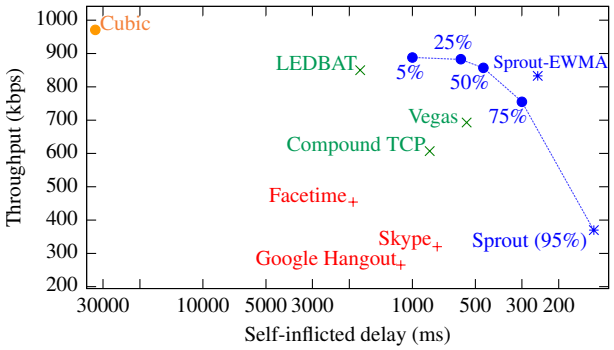
Although it is purely end-to-end, Sprout's delays are even lower than Cubic-over-CoDel. However, this comes at a cost to throughput. (Figures are given in the table in the introduction.) Sprout-EWMA achieves within 6% of the same delay as Cubic-over-CoDel, with 30% more throughput.

Rather than embed a single throughput-delay tradeoff into the network (e.g. by installing CoDel on carrier infrastructure), we believe it makes architectural sense to provide endpoints and applications with such control when possible. Users should be able to decide which throughput-delay compromise they prefer. In this setting, it appears achievable to match or even exceed CoDel's performance without modifying gateways.

5.5 Effect of confidence parameter

The Sprout receiver makes forecasts of a lower bound on how many packets will be delivered with at least 95% probability. We explored the effect of lowering this confidence parameter to express a greater willingness that

Figure 9: Lowering the forecast's confidence parameter allows greater throughput at the cost of more delay. Results on the T-Mobile 3G (UMTS) uplink:



packets be queued for longer than the sender's 100 ms tolerance.

Results on one network path are shown in Figure 9. The different confidence parameters trace out a curve of achievable throughput-delay tradeoffs. As expected, decreasing the amount of caution in the forecast allows the sender to achieve greater throughput, but at the cost of more delay. Interestingly, although Sprout achieves higher throughput and lower delay than Sprout-EWMA by varying the confidence parameter, it never achieves both at the same time. Why this is—and whether Sprout's stochastic model can be further improved to beat Sprout-EWMA simultaneously on both metrics—will need to be the subject of further study.

5.6 Loss resilience

The cellular networks we experimented with all exhibited low packet loss rates, but that will not be true in general. To investigate the loss resilience of Sprout, we used the traces collected from one network (Verizon LTE) and simulated Bernoulli packet losses (tail drop) with two different packet loss probabilities, 5% and 10% (in each direction). The results are shown in the table below:

Protocol	Throughput (kbps)	Delay (ms)
Downlink		
Sprout	4741	73
Sprout-5%	3971	60
Sprout-10%	2768	58
Uplink		
Sprout	3703	332
Sprout-5%	2598	378
Sprout-10%	1163	314

As expected, the throughput does diminish in the face of packet loss, but Sprout continues to provide good throughput even at high loss rates. (TCP, which interprets loss as a congestion signal, generally suffers unacceptable slowdowns in the face of 10% each-way packet loss.) These results demonstrate that Sprout is relatively resilient to packet loss.

5.7 Sprout as a tunnel for competing traffic

We tested whether SproutTunnel, used as a tunnel over the cellular link to a well-connected relay, can successfully isolate bulk-transfer downloads from interactive applications.

We ran two flows: a TCP Cubic bulk transfer (download only) and a two-way Skype videoconference, using the Linux version of Skype.

We compared the situation of these two flows running directly over the emulated Verizon LTE link, versus running them through SproutTunnel over the same link. The experiments lasted about ten minutes each.[9]

	Direct	via Sprout	Change
Cubic throughput	8336 kbps	3776 kbps	−55%
Skype throughput	78 kbps	490 kbps	+528%
Skype 95% delay	6.0 s	0.17 s	−97%

The results suggest that interactive applications can be greatly aided by having their traffic run through Sprout along with bulk transfers. Without Sprout to mediate, Cubic squeezes out Skype and builds up huge delays. However, Sprout's conservatism about delay also imposes a substantial penalty to Cubic's throughput.

6 Related Work

End-to-end algorithms. Traditional congestion-control algorithms generally do not simultaneously achieve high utilization and low delay over paths with high rate variations. Early TCP variants such as Tahoe and Reno [10] do not explicitly adapt to delay (other than from ACK clocking), and require an appropriate buffer size for good performance. TCP Vegas [4], FAST [12], and Compound TCP [20] incorporate round-trip delay explicitly, but the adaptation is reactive and does not directly involve the receiver's observed rate.

LEDBAT [19] (and TCP Nice [21]) share our goals of high throughput without introducing long delays, but LEDBAT does not perform as well as Sprout. We believe this is because of its choice of congestion signal (one-way delay) and the absence of forecasting. Some recent work proposes TCP receiver modifications to combat bufferbloat in 3G/4G wireless networks [11]. Schemes such as "TCP-friendly" equation-based rate control [7] and binomial congestion control [1] exhibit slower transmission rate variations than TCP, and in principle could introduce lower delay, but perform poorly in the face of sudden rate changes [2].

[9] In each run, Skype ended the video portion of the call once and was restarted manually.

Google has proposed a congestion-control scheme [15] for the WebRTC system that uses an arrival-time filter at the receiver, along with other congestion signals, to decide whether a real-time flow should increase, decrease, or hold its current bit rate. We plan to investigate this system and assess it on the same metrics as the other schemes in our evaluation.

Active queue management. Active queue management schemes such as RED [8] and its variants, BLUE [6], AVQ [14], etc., drop or mark packets using local indications of upcoming congestion at a bottleneck queue, with the idea that endpoints react to these signals before queues grow significantly. Over the past several years, it has proven difficult to automatically configure the parameters used in these algorithms. To alleviate this shortcoming, CoDel [17] changes the signal of congestion from queue length to the delay experienced by packets in a queue, with a view toward controlling that delay, especially in networks with deep queues ("bufferbloat" [9]).

Our results show that Sprout largely holds its own with CoDel over challenging wireless conditions without requiring any gateway modifications. It is important to note that network paths in practice have several places where queues may build up (in LTE infrastructure, in baseband modems, in IP-layer queues, near the USB interface in tethering mode, etc.), so one may need to deploy CoDel at all these locations, which could be difficult. However, in networks where there is a lower degree of isolation between queues than the cellular networks we study, CoDel may be the right approach to controlling delay while providing good throughput, but it is a "one-size-fits-all" method that assumes that a single throughput-delay tradeoff is right for all traffic.

7 Limitations and Future Work

Although our results are encouraging, there are several limitations to our work. First, as noted in §2 and §3, an end-to-end system like Sprout cannot control delays when the bottleneck link includes competing traffic that shares the same queue. If a device uses traditional TCP outside of Sprout, the incurred queueing delay—seen by Sprout and every flow—will be substantial.

Sprout is not a traditional congestion-control protocol, in that it is designed to adapt to varying link conditions, not varying cross traffic. In a cellular link where users have their own queues on the base station, interactive performance will likely be best when the user runs bulk and interactive traffic *inside* Sprout (e.g. using SproutTunnel), not alongside Sprout. We have not evaluated the performance of multiple Sprouts sharing a queue.

The accuracy of Sprout's forecasts depends on whether the application is providing offered load sufficient to saturate the link. For applications that switch intermittently on and off, or don't desire high throughput, the transient behavior of Sprout's forecasts (e.g. ramp-up time) becomes more important. We did not evaluate any non-saturating applications in this paper or attempt to measure or optimize Sprout's startup time from idle.

Finally, we have tested Sprout only in trace-based emulation of eight cellular links recorded in the Boston area in 2012. Although Sprout's model was frozen before data were collected and was not "tuned" in response to any particular network, we cannot know how generalizable Sprout's algorithm is without more real-world testing.

In future work, we are eager to explore different stochastic network models, including ones trained on empirical variations in cellular link speed, to see whether it is possible to perform much better than Sprout if a protocol has more accurate forecasts. We think it will be worthwhile to collect enough traces to compile a standardized benchmark of cellular link behavior, over which one could evaluate any new transport protocol.

8 Conclusion

This paper presented Sprout, a transport protocol for real-time interactive applications over Internet paths that traverse cellular wireless networks. Sprout improves on the performance of current approaches by modeling varying networks explicitly. Sprout has two interesting ideas: the use of packet arrival times as a congestion signal, and the use of probabilistic inference to make a cautious forecast of packet deliveries, which the sender uses to pace its transmissions. Our experiments show that forecasting is important to controlling delay, providing an end-to-end rate control algorithm that can react at time scales shorter than a round-trip time.

Our experiments conducted on traces from four commercial cellular networks show many-fold reductions in delay, and increases in throughput, over Skype, Facetime, and Hangout, as well as over Cubic, Compound TCP, Vegas, and LEDBAT. Although Sprout is an end-to-end scheme, in this setting it matched or exceeded the performance of Cubic-over-CoDel, which requires modifications to network infrastructure to be deployed.

9 Acknowledgments

We thank Shuo Deng, Jonathan Perry, Katrina LaCurts, Andrew McGregor, Tim Shepard, Dave Täht, Michael Welzl, Hannes Tschofenig, and the anonymous reviewers for helpful discussion and feedback. This work was supported in part by NSF grant 1040072. KW was supported by the Claude E. Shannon Research Assistantship. We thank the members of the MIT Center for Wireless Networks and Mobile Computing, including Amazon.com, Cisco, Google, Intel, Mediatek, Microsoft, ST Microelectronics, and Telefonica, for their support.

REFERENCES

[1] D. Bansal and H. Balakrishnan. Binomial Congestion Control Algorithms. In *INFOCOM*, 2001.

[2] D. Bansal, H. Balakrishnan, S. Floyd, and S. Shenker. Dynamic Behavior of Slowly-Responsive Congestion Control Algorithms. In *SIGCOMM*, 2001.

[3] P. Bender, P. Black, M. Grob, R. Padovani, N. Sindhushyana, and A. Viterbi. A bandwidth efficient high speed wireless data service for nomadic users. *IEEE Communications Magazine*, July 2000.

[4] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM*, 1994.

[5] D. Cox. Long-range dependence: A review. In *H.A. David and H.T. David, editors, Statistics: An Appraisal*, pages 55–74. Iowa State University Press, 1984.

[6] W. Feng, K. Shin, D. Kandlur, and D. Saha. The BLUE active queue management algorithms. *IEEE/ACM Trans. on Networking*, Aug. 2002.

[7] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-Based Congestion Control for Unicast Applications. In *SIGCOMM*, 2000.

[8] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. on Networking*, 1(4), Aug. 1993.

[9] J. Gettys and K. Nichols. Bufferbloat: Dark buffers in the internet. *Queue*, 9(11):40:40–40:54, Nov. 2011.

[10] V. Jacobson. Congestion Avoidance and Control. In *SIGCOMM*, 1988.

[11] H. Jiang, Y. Wang, K. Lee, and I. Rhee. Tackling bufferbloat in 3g/4g mobile networks. *NCSU Technical report*, March 2012.

[12] C. Jin, D. Wei, and S. Low. Fast tcp: motivation, architecture, algorithms, performance. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2490 – 2501 vol.4, march 2004.

[13] S. Keshav. Packet-Pair Flow Control. *IEEE/ACM Trans. on Networking*, Feb. 1995.

[14] S. Kunniyur and R. Srikanth. Analysis and design of an adaptive virtual queue (AVQ) algorithm for active queue management. In *SIGCOMM*, 2001.

[15] H. Lundin, S. Holmer, and H. Alvestrand. A Google Congestion Control Algorithm for Real-Time Communication on the World Wide Web. http://tools.ietf.org/html/draft-alvestrand-rtcweb-congestion-03, 2012.

[16] R. Mahajan, J. Padhye, S. Agarwal, and B. Zill. High performance vehicular connectivity with opportunistic erasure coding. In *USENIX*, 2012.

[17] K. Nichols and V. Jacobson. Controlling queue delay. *ACM Queue*, 10(5), May 2012.

[18] V. Paxson and S. Floyd. Wide-Area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Trans. on Networking*, 3(3):226–244, June 1995.

[19] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low Extra Delay Background Transport (LEDBAT), 2012. IETF RFC 6817.

[20] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-Speed and Long Distance Networks. In *INFOCOM*, 2006.

[21] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: a mechanism for background transfers. *SIGOPS Oper. Syst. Rev.*, 36(SI):329–343, Dec. 2002.

Demystifying Page Load Performance with WProf

Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall
University of Washington

Abstract

Web page load time is a key performance metric that many techniques aim to reduce. Unfortunately, the complexity of modern Web pages makes it difficult to identify performance bottlenecks. We present WProf, a lightweight in-browser profiler that produces a detailed dependency graph of the activities that make up a page load. WProf is based on a model we developed to capture the constraints between network load, page parsing, JavaScript/CSS evaluation, and rendering activity in popular browsers. We combine WProf reports with critical path analysis to study the page load time of 350 Web pages under a variety of settings including the use of end-host caching, SPDY instead of HTTP, and the mod_pagespeed server extension. We find that computation is a significant factor that makes up as much as 35% of the critical path, and that synchronous JavaScript plays a significant role in page load time by blocking HTML parsing. Caching reduces page load time, but the reduction is not proportional to the number of cached objects, because most object loads are not on the critical path. SPDY reduces page load time only for networks with high RTTs and mod_pagespeed helps little on an average page.

1 Introduction

Web pages delivered by HTTP have become the de-facto standard for connecting billions of users to Internet applications. As a consequence, Web page load time (PLT) has become a key performance metric. Numerous studies and media articles report its importance for user experience [5, 4], and consequently to business revenues. For example, Amazon increased revenue 1% for every 0.1 second reduction in PLT, and Shopzilla experienced a 12% increase in revenue by reducing PLT from 6 seconds to 1.2 seconds [23].

Given its importance, many techniques have been developed and applied to reduce PLT. They range from caching and CDNs, to more recent innovations such as the SPDY protocol [28] that replaces HTTP, and the mod_pagespeed server extension [19] to re-write pages. Other proposals include DNS pre-resolution [9], TCP pre-connect [30], Silo [18], TCP fast open [24], and ASAP [35].

Thus it is surprising to realize that the performance bottlenecks that limit the PLT of modern Web pages are still not well understood. Part of the culprit is the complexity of the page load process. Web pages mix resources fetched by HTTP with JavaScript and CSS evaluation. These activities are inter-related such that the bottlenecks are difficult to identify. Web browsers complicate the situation with implementation strategies for parsing, loading and rendering that significantly impact PLT. The result is that we are not able to explain why a change in the way a page is written or how it is loaded has an observed effect on PLT. As a consequence, it is difficult to know when proposed techniques will help or harm PLT.

Previous studies have measured Web performance in different settings, e.g., cellular versus wired [13], and correlated PLT with variables such as the number of resources and domains [7]. However, these factors are only coarse indicators of performance and lack explanatory power. The key information that can explain performance is the dependencies within the page load process itself. Earlier work such as WebProphet [16] made clever use of inference techniques to identify some of these dependencies. But inference is necessarily time-consuming and imprecise because it treats the browser as a black-box. Most recently, many "waterfall" tools such as Google's Pagespeed Insight [22] have proliferated to provide detailed and valuable timing information on the components of a page load. However, even these tools are limited to reporting what happened without explaining why the page load proceeded as it did.

Our goal is to demystify Web page load performance. To this end, we abstract the dependency policies in four browsers, i.e., IE, Firefox, Chrome, and Safari. We run experiments with systematically instrumented test pages and observe object timings using Developer Tools [8]. For cases when Developer Tools are insufficient, we deduce dependencies by inspecting the browser code when open source code is available. We find that some of these dependency policies are given by Web standards, e.g., JavaScript evaluation in script tags blocks HTML parsing. However, other dependencies are the result of browser implementation choices, e.g., a single thread of execution shared by parsing, JavaScript and CSS evaluation, and rendering. They have significant impacts on PLT and cannot be ignored.

Given the dependency policies, we develop a lightweight profiler, WProf, that runs in Webkit browsers (e.g., Chrome, Safari) while real pages are loaded. WProf generates a dependency graph and identifies a load bottleneck for any given Web page. Unlike existing tools that produce waterfall or HAR reports [12], our

profiler discovers and reports the dependencies between the browser activities that make up a page load. It is this information that pinpoints why, for example, page parsing took unexpectedly long and hence suggests what may be done to improve PLT.

To study page load performance, we run WProf while fetching pages from popular servers and apply critical path analysis, a well-known technique for analyzing the performance of parallel programs [25]. First, we identify page load bottlenecks by computing what fraction of the critical path the activity occupies. Surprisingly, we find that while most prior work focuses on network activity, computation (mostly HTML parsing and JavaScript execution) comprises 35% of the critical path. Interestingly, downloading HTML and synchronous JavaScript (which blocks parsing) makes up a large fraction of the critical path, while fetching CSS and asynchronous JavaScript makes up little of the critical path. Second, we study the effectiveness of different techniques for optimizing web page load. Caching reduces the volume of data substantially, but decreases PLT by a lesser amount because many of the downloads that benefit from it are not on the critical path. Disappointingly, SPDY makes little difference to PLT under its default settings and low RTTs because it trades TCP connection setup time for HTTP request sending time and does not otherwise change page structure. Mod_pagespeed also does little to reduce PLT because minifying and merging objects does not reduce network time on critical paths.

We make three contributions in this paper. The first is our activity dependency model of page loads, which captures the constraints under which real browsers load Web pages. The second contribution is WProf, an in-browser profiling tool that records page load dependencies and timings with minimal runtime overhead. Our third contribution is the study of extensive page loads that uses critical path analysis to identify bottlenecks and explain the limited benefits of SPDY and mod_pagespeed.

In the rest of this paper, we describe the page load process (§2) and our activity dependency model (§3). We then describe the design and implementation of WProf (§4). We use WProf for page load studies (§5) before presenting related work (§6) and concluding (§7).

2 Background

We first provide background on how browsers load Web pages. Figure 1 shows the workflow for loading a page. The page load starts with a user-initiated request that triggers the *Object Loader* to download the corresponding root HTML page. Upon receiving the first chunk of the root page, the *HTML Parser* starts to iteratively parse the page and download embedded objects within the page, until the page is fully parsed. The embedded objects are *Evaluated* when needed. To visualize

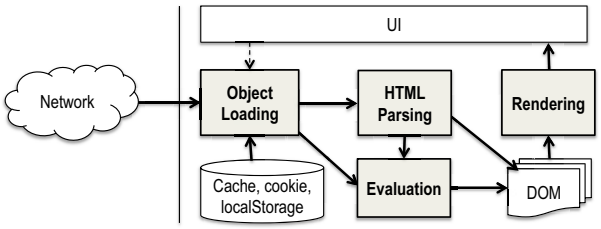


Figure 1: The workflow of a page load. It involves four processes (shown in gray).

the page, the *Rendering Engine* progressively renders the page on the browser. While the HTML Parser, Evaluator, and Rendering Engine are computation processes, the Object Loader is a network process.

HTML Parser: The Parser is key to the page load process, and it transforms the raw HTML page to a document object model (DOM) tree. A DOM tree is an intermediate representation of a Web page; the nodes in the DOM tree represent HTML tags, and each node is associated with a set of attributes. The DOM tree representation provides a common interface for programs to manipulate the page.

Object Loader: The Loader fetches objects requested by the user or those embedded in the HTML page. The objects are fetched over the Internet using HTTP or SPDY [28], unless the objects are already present in the browser cache. The embedded objects fall under different mime types: HTML (e.g., IFrame), JavaScript, CSS, Image, and Media. Embedded HTMLs are processed separately and use a different DOM tree. Inlined JavaScript and inlined CSS do not need to be loaded.

Evaluator: Two of the five embedded object types, namely, JavaScript and CSS, require additional evaluation after being fetched. JavaScript is a piece of software that adds dynamic content to Web pages. Evaluating JavaScript involves manipulating the DOM, e.g., adding new nodes, modifying existing nodes, or changing nodes' styles. Since both JavaScript Evaluation and HTML Parsing modify the DOM, HTML parsing is blocked for JavaScript evaluation to avoid conflicts in DOM modification. However, when JavaScript is tagged with an *async* attribute, the JavaScript can be downloaded and evaluated in the background without blocking HTML Parsing. In the rest of the paper, JavaScript refers to synchronous JavaScript unless stated.

Cascading style sheets (CSS) are used for specifying the presentational attributes (e.g., colors and fonts) of the HTML content and is expressed as a set of rules. Evaluating a CSS rule involves changing styles of DOM nodes. For example, if a CSS rule specifies that certain nodes are to be displayed in blue color, then evaluating the CSS involves identifying the matching nodes in the DOM (known as CSS selector matching) and adding the

style to each matched node. JavaScript and CSS are commonly embedded in Web pages today [7].

Rendering engine: Browsers render Web pages progressively as the HTML Parser builds the DOM tree. Rendering involves two processes–Layout and Painting. Layout converts the DOM tree to the layout tree that encodes the size and position of each visible DOM node. Painting converts this layout tree to pixels on the screen.

3 Browser Dependency Policies

Web page dependencies are caused by various factors such as co-dependence between the network and the computation activities, manipulation of common objects, limited resources, etc. Browsers use various policies to enforce these dependencies. Our goal is to abstract the browser dependency policies. We use this policy abstraction to efficiently extract the dependency structure of any given Web page (§4).

3.1 Dependency definition

Ideally, the four processes involved in a page load (described in Figure 1) would be executed in parallel, so that the page load performance is determined by the slowest process. However, the processes are inter-dependent and often block each other. To represent the dependency policies, we first discretize the steps involved in each process. The granularity of discretization should be fine enough to reflect dependencies and coarse enough to preserve semantics. We consider the most coarse-grained atomic unit of work, which we call an *activity*. In the case of HTML Parser, the activity is parsing a single tag. The Parser repeatedly executes this activity until all tags are parsed. Similarly, the Object Loader activity is loading a single object, the Evaluator activity is evaluating a single JavaScript or CSS, and the activity of the Rendering process is rendering the current DOM.

We say an activity a_i is *dependent* on a previously scheduled activity a_j, if a_i can be executed only after a_j has completed. There are two exceptions to this definition, where the activity is executed after only a *partial* completion of the previous activity. We discuss these exceptions in §3.3.

3.2 Methodology

We extract browser dependency policies by (i) inspecting browser documentation, (ii) inspecting browser code if open-source code is available, and (iii) systematically instrumenting test pages. Note that no single method provides a complete set of browser policies, but they complement each other in the information they provide. Our methodology assumes that browsers tend to parse tags sequentially. However, one exception is *preloading*. Preloading means that the browser preemptively loads objects that are embedded later in the page, even before their corresponding tags are parsed. Preloading is often used to speed up page loads.

Below, we describe how we instrument and experiment with test pages. We conduct experiments in four browsers: Chrome, Firefox, Internet Explorer, and Safari. We host our test pages on controlled servers. We observe the load timings of each object in the page using Developer Tools [8] made available by the browsers. We are unable to infer dependencies such as rendering using Developer Tools. Instead, for open-source browsers, we inspect browser code to study the dependency policies associated with rendering.

Instrumenting test pages (network): We instrument test pages to exhaustively cover possible loading scenarios: (i) loading objects in different orders, and (ii) loading embedded objects. We list our instrumented test pages and results at `wprof.cs.washington.edu/tests`. Web pages can embed five kinds of objects as described in §2. We first create test pages that embed all combinations of object pairs, e.g., the test page may request an embedded JavaScript followed by an image. Next, we create test pages that embed more than two objects in all possible combinations. Embedded objects may further embed other objects. We create test pages for each object type that in-turn embeds all combinations of objects. To infer dependency policies, we systematically inject delays to objects and observe load times of other objects to see whether they are delayed accordingly, similar to the technique used in WebProphet [16]. For example, in a test page that embeds a JavaScript followed by an image, we delay loading the Javascript and observe whether the image is delayed.

Instrumenting test pages (computation): Developer tools expose timings of network activities, but not timings of computational activities. We instrument test pages to circumvent this problem and study dependencies during two main computational activities: HTML parsing and JavaScript evaluation. HTML parsing is often blocked during page load. To study the blocking behavior across browsers, we create test pages for each object type. For each page, we also embed an IFrame in the end. During page load, if the IFrame begins loading only after the previous object finishes loading, we infer that HTML parsing is blocked during the object load. IFrames are ideal for this purpose because they are not preloaded by browsers. To identify dependencies related to JavaScript evaluation, we create test pages that contain scripts with increasing complexity; i.e., scripts that require more and more time for evaluation. We embed an IFrame at the end. As before, if IFrame does not load immediately after the script loads, we infer that HTML parsing is blocked for script evaluation.

Dependency	Name	Definition
Flow	F1	Loading an object → Parsing the tag that references the object
	F2	Evaluating an object → Loading the object
	F3	Parsing the HTML page → Loading the first block of the HTML page*
	F4	Rendering the DOM tree → Updating the DOM
	F5	Loading an object referenced by a JavaScript or CSS → Evaluating the JavaScript or CSS*
	F6	Downloading/Evaluating an object → Listener triggers or timers
Output	O1	Parsing the next tag → Completion of a previous JavaScript download and evaluation
	O2	JavaScript evaluation → Completion of a previous CSS evaluation
	O3	Parsing the next tag → Completion of a previous CSS download and evaluation
Lazy/Eager binding	B1	[Lazy] Loading an image appeared in a CSS → Parsing the tag decorated by the image
	B2	[Lazy] Loading an image appeared in a CSS → Evaluation of any CSS that appears in front of the tag decorated by the image
	B3	[Eager] Preloading embedded objects does not depend on the status of HTML parsing. (breaks F1)
Resource constraint	R1	Number of objects fetched from different servers → Number of TCP connections allowed per domain
	R2	Browsers may execute key computational activities on the same thread, creating dependencies among the activities.This dependency is determined by the scheduling policy.

* An activity depends on *partial* completion of another activity.

Table 1: Summary of dependency policies imposed by browsers. → represents "depends on" relationship.

3.3 Dependency policies

Using our methodology, we uncover the dependency policies in browsers and categorize them as: Flow dependency, Output dependency, Lazy/Eager binding dependency, and dependencies imposed by resource constraints. Table 1 tabulates the dependency policies. While output dependencies are required for correctness, the dependencies imposed by lazy/eager binding and resource constraints are a result of browser implementation strategies.

Flow dependency is the simplest form of dependency. For example, loading an object depends on parsing a tag that references the object (F1). Similarly, evaluating a JavaScript depends on loading the JavaScript (F2). Often, browsers may load and evaluate a JavaScript based on triggers and timeouts, rather than the content of the page (F6). Table 1 provides the complete set of flow dependencies. Note that dependencies F3 and F5 are special cases, where the activity only depends on the partial completion of the previous activity. In case of F3, the browser starts to parse the page when the first chunk of the page is loaded, not waiting for the entire load to be completed. In case of F5, an object requested by a JavaScript/CSS is loaded immediately after evaluation starts, not waiting for the evaluation to be completed.

Output dependency ensures the correctness of execution when multiple processes modify a shared resource and execution order matters. In browsers, the shared resource is the DOM. Since both JavaScript evaluation and HTML parsing may write to the DOM, HTML parsing is blocked until JavaScript is both loaded and evaluated (O1). This ensures that the DOM is modified in the order specified in the page. Since JavaScript can modify styles of DOM nodes, execution of JavaScript waits for the completion of CSS processing (O2). Note that *async* JavaScript is not bounded by output dependencies because the order of script execution does not matter.

Lazy/Eager binding: Several lazy/eager bindings techniques are used by the browser to trade off between decreasing spurious downloads and improving latency. Preloading (B3) is an example of an eager binding technique where browsers preemptively load objects that are embedded later in the page. Dependency B1 is a result of a lazy binding technique. When a CSS object is downloaded and evaluated, it may include an embedded image, for example, to decorate the background or to make CSS sprites. The browser does not load this image as soon as CSS is evaluated, and instead waits until it parses a tag that is decorated by the image. This ensures that the image is downloaded only if it is used.

Resource constraints: Browsers constrain the use of two resources—compute power and network resource. With respect to network resources, browsers limit the number of TCP connections. For example, Firefox limits the number of open TCP connections per domain to 6 by default. If a page load process needs to load more than 6 embedded objects from the same domain simultaneously, the upcoming load is blocked until a previous load completes. Similarly, some browsers allocate a single compute thread to execute certain computational activities. For example, WebKit executes parts of rendering in the parsing thread. This results in blocking parsing until rendering is complete (R2). We were able to observe this only for the open-source WebKit browser because

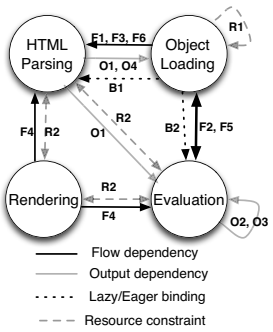


Figure 2: Dependencies between processes.

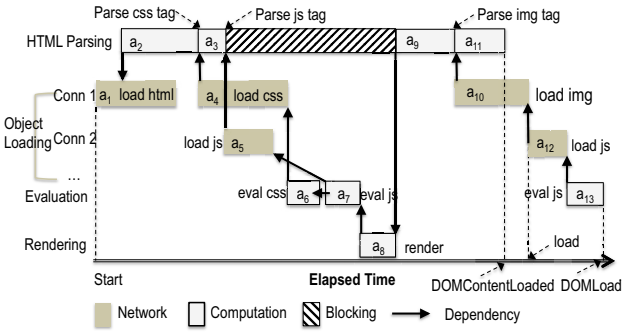


Figure 3: Dependency graph. Arrows denote"depends on" relation and vertices represent activities in a page load.

```
<html>
  <head>
    <link href='a.css'
      rel='stylesheet'>
    <script src='b.js' />
  </head>
  <!-- req a JS -->
  <body onload='...'>
    <img src='c.png' />
  </body>
</html>
```

Figure 4: Corresponding example code.

Dependency	IE	Firefox	WebKit
Output	all	no O3	no O3
Late binding	all	all	all
Eager	*Preloads	Preloads	Preloads
Binding	img, JS, CSS	img, JS, CSS	JS, CSS
Resource (R1)	6 conn.	6 conn.	6 conn.

Table 2: Dependency policies across browsers.

we directly instrumented the WebKit engine to log precise timing information of computational activities.

Figure 2 summarizes how these dependencies affect the four processes. Note that more than one dependency relationship can exist between two activities. For example, consider a page containing a CSS object followed by a JavaScript object. Evaluating the JavaScript depends on both loading the JavaScript (F2) and evaluating the previously appearing CSS (O3). The timing of these two dependencies will determine which of the two dependencies occur in this instance.

3.4 Dependency policies across browsers

Table 2 show the dependency policies across browsers. Only IE enforces dependency O3 that blocks HTML parsing to download and evaluate CSS. The preloading policies (i.e., when and what objects to preload) also differ among browsers, while we note that no browser preloads embedded IFrames. Note that flow dependency is implicitly imposed by all browsers. We were unable to compare the compute dependency (R2) across browsers because it requires modification to the browser code.

3.5 Dependency graph of an example page

Figure 3 shows a dependency graph of an example page. The `DOMContentLoaded` refers to the event that HTML finishes parsing, `load` refers to the event that all embedded objects are loaded, and `DOMLoad` refers to the event that DOM is fully loaded. Our example Web page (Figure 4) contains an embedded CSS, a JavaScript, an image, and a JavaScript triggered on the `load` event. Many Web pages (e.g., `facebook.com`) may load additional objects on the `load` event fires.

The page load starts with loading the root HTML page (a_1), following which parsing begins (a_2). When the Parser encounters the CSS tag, it loads the CSS (a_4) but does not block parsing. However, when the Parser encounters the JavaScript tag, it loads the JavaScript (a_5) and blocks parsing. Note that loading the CSS and the JavaScript subjects to a resource constraint; i.e., both CSS and JavaScript can be loaded simultaneously only if multiple TCP connections can be opened per domain. CSS is evaluated (a_6) after being loaded. Even though JavaScript finishes loading, it needs to wait until the CSS finishes evaluating, and then the JavaScript is evaluated (a_7). The rendering engine renders the current DOM (a_8) after which HTML parsing resumes (a_9). The Parser then encounters and loads an image (a_{10}) after which HTML parsing is completed and fires a `load` event. In our example, the triggered JavaScript is loaded (a_{12}) and evaluated (a_{13}). When all embedded objects are evaluated and the DOM is updated, the `DOMLoad` event is fired. Even our simple Web page exhibits over 10 dependencies of different types. For example, $a_2 \rightarrow a_1$ is a flow dependency; $a_7 \rightarrow a_6$ is an output dependency; and $a_9 \rightarrow a_8$ is a resource dependency.

The figure also illustrates the importance of using dependency graphs for fine-grained accounting of page load time. For example, both activity a_4 and a_5 are network loads. However, only activity a_4 contributes to page load time; i.e., decreasing the load time of a_5 does not decrease total page load time. Tools such as HTTP Archival Records [12] provide network time for each activity, but this cannot be used to isolate the bottleneck activities. The next section demonstrates how to isolate the bottleneck activities using dependency graphs.

4 WProf

We present WProf, a tool that captures the dependency graph for any given Web page and identifies the delay bottlenecks. Figure 5 shows WProf architecture. The primary component is an in-browser profiler that instruments the browser engine to obtain timing and depen-

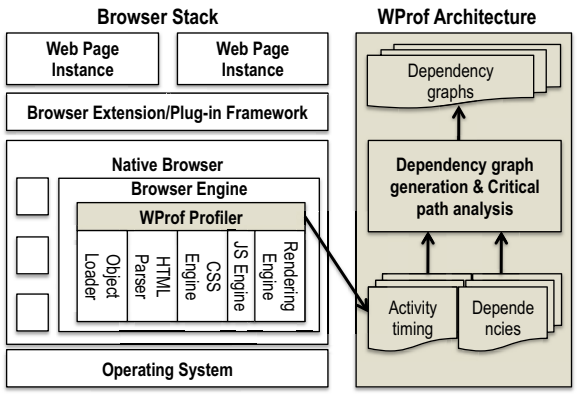


Figure 5: The WProf Architecture. WProf operates just above the browser engine, allowing it to collect precise timing and dependency information.

dency information at runtime. The profiler is a shim layer inside the browser engine and requires minimal changes to the rest of the browser. Note that we do not work at the browser extension or plugin level because they provide limited visibility into the browser internals. WProf then analyzes the profiler logs offline to generate the dependency graphs and identify the bottleneck path. The profiler is lightweight and has negligible effect on page load performance (§4.4).

4.1 WProf Profiler

The key role of WProf profiler is to record timing and dependency information for a page load. While the dependency information represents the structure of a page, the timing information captures how dependencies are exhibited for a specific page load; both are crucial to pinpoint the bottleneck path.

Logging Timing: WProf records the timestamps at the beginning and end of each activity executed during the page load process. WProf also records network timing information, including DNS lookup, TCP connection setup, and HTTP transfer time. To keep track of the number of TCP connections being used/re-used, WProf records the IDs of all TCP connections.

Logging dependencies: WProf assumes the dependency policies described in §3 and uses different ways to log different kinds of dependencies. Flow dependencies are logged by attaching the URL to an activity. For example in Figure 3, WProf learns that the activity that evaluates b.js depends on the activity that loads b.js by recording b.js. WProf logs resource constraints by IDs of the constrained resources, e.g., thread IDs and TCP IDs.

To record output dependencies and lazy/eager bindings, WProf tracks a DOM-specific ordered sequence of processed HTML tags as the browser loads a page. We maintain a separate ordered list for each DOM tree associated with the page (e.g., the DOM for the root page and the various IFrames). HTML tags are recorded when they are first encountered; they are then attached to the activities that occur when the tags are being parsed. For example, when objects are preloaded, the tags under parsing are attached to the preloading activity, not the tags that reference the objects. For example in Figure 4, the Parser first processes the tag that references a.css, and then processes the tag that references b.js. This ordering, combined with the knowledge of output dependencies, result in the dependency $a_7 \rightarrow a_6$ in Figure 3. Note that the HTML page itself provides an implicit ordering of the page activities; however, this ordering is static. For example, if a JavaScript in a page modifies the rest of the page, statically analyzing the Web page provides an incorrect order sequence. Instead, WProf records the Web page processing order directly from the browser runtime.

Validating the completeness of our dependency policies would require either reading all the browser code or visiting all the Web pages, neither of which is practical. As browsers evolve constantly, changes in Web standard or browser implementations can change the dependency policies. Thus, WProf needs to be modified accordingly. Since WProf works as a shim layer in browsers that does not require knowledge about underlying components such as CSS evaluation, the effort to record an additional dependency policy would be minimal.

4.2 Critical path analysis

To identify page load bottlenecks, we apply critical path analysis to dependency graphs. Let the dependency graph $G = (V, E)$, where V is the set of activities required to render the page and E is the set of dependencies between the activities. Each activity $a \in V$ is associated with the attribute that represents the duration of the activity. The critical path P consists of a set of activities that form the longest path in the dependency graph such that, reducing the duration of any activity *not* on the critical path ($a \notin P$), will not change the critical path; i.e., optimizing an activity not on the critical path will not reduce page load performance. For example, the critical path for the dependency graph shown in Figure 3 is (a_1, a_2, a_4, a_6, a_7, a_8, a_9, a_{11}, a_{12}, a_{13}).

We estimate the critical path P of a dependency graph using the algorithm below. We treat preloading as a special case because it breaks the flow dependency. When preloading occurs, we always add it to the critical path.

4.3 Implementation

We implement WProf on Webkit [33], an open source Web browser engine that is used by Chrome, Safari, Android, and iOS. The challenge is to ensure that the WProf shim layer is lightweight. Creating a unique identifier for each activity (for logging dependency relationship) is memory intensive as typical pages require several thou-

$P \leftarrow \emptyset$; $a \leftarrow$ Activity that completes last;
$P \leftarrow P \cup a$;
while *a is not the first activity* **do**
  $A \leftarrow$ Set of activities that a is dependent on;
  if *a is preloaded from an activity a' in A* **then**
    $a \leftarrow a'$;
  else
    $a \leftarrow$ Activity in A that completes last;
  end
  $P \leftarrow P \cup a$;
end
return P

WProf	CPU %	Memory %
on	58.5	65.5
off	53.5	54.9

Table 3: Maximum sampled CPU and memory usage.

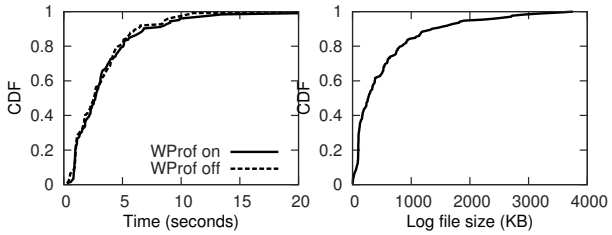
sands of unique identifiers (e.g., for HTML tags). Instead, we use pointer addresses as unique identifiers. The WProf profiler keeps all logs in memory and transfers them to disk after the DOM is loaded, to minimize the impact on page load.

We extended Chrome and Safari by modifying 2K lines of C++ code. Our implementation can be easily extended to other browsers that build on top of Webkit. Our dependency graph generation and critical path analysis is written in Perl and PHP. The WProf source code is available at `wprof.cs.washington.edu`.

4.4 System evaluation

In this section, we present WProf micro-benchmarks. We perform the experiments on Chrome. We use a 2GHz CPU dual core and 4GB memory machine running MacOS. We load 200 pages with a five second interval between pages. The computer is connected via Ethernet, and to provide a stable network environment, we limit the bandwidth to 10Mbps using DummyNet [10].

Figure 6(a) shows the CDF of page load times with and without WProf. We define page load time as the time from when the page is requested to when the `DOMLoad` (Figure 3) event is fired. We discuss our rationale for the page load time definition in §5.1. The figure shows that WProf's in-browser profiler only has a negligible effect on the page load time. Similarly, Figure 6(b) shows the CDF of size of the logs generated by the WProf profiler. The median log file size is 268KB even without compression, and is unlikely to be a burden on the storage system. We sample the CPU and memory usage at a rate of 0.1 second when loading the top 200 web pages with and without WProf. Table 3 shows that even in the maximum case, WProf only increases the CPU usage by 9.3% and memory usage by 19.3%.



(a) CDF of page load time with and without WProf.

(b) CDF of log file sizes.

Figure 6: WProf evaluation.

5 Studies with WProf

The goal of our studies is to use WProf's dependency graph and critical path analysis to (i) identify the bottleneck activities during page load (§5.2), (ii) quantify page load performance under caching (§5.3), and (iii) quantify page load performance under two proposed optimization techniques, SPDY and mod_pagespeed (§5.4).

5.1 Methodology

Experimental setup: We conduct our experiments on default Chrome and WProf-instrumented Chrome. We automate our experiments using the Selenium Webdriver [27] that emulates browser clicks. By default, we present experiments conducted on an iMac with a 3GHz quad core CPU and 8GB memory. The computer is connected to campus Ethernet at UW Seattle. We use DummyNet [10] to provide a stable network connection of 10Mbps bandwidth; 10Mbps represents the average broadband bandwidth seen by urban users [6]. By default, we assume page loads are *cold*, i.e., the local cache is cleared. This is the common case, as 40%–60% of page loads are known to be cold loads [18]. We report the minimum page load time from a total of 5 runs.

Web pages: We experiment with the top 200 most visited websites from Alexa [3]. Because some websites get stuck due to reported hang bugs in Selenium Webdriver, we present results from the 150 websites that provide complete runs.

Page load time metric: We define Page Load Time (PLT) as the time between when the page is requested and when the `DOMLoad` event is fired. Recall that the `DOMLoad` event is fired when all embedded objects are fetched and added to the DOM (see Figure 3). We obtain all times directly from the browser engine. There are a few alternative definitions to PLT. The *above-the-fold* time [1] metric is a user-driven metric that measures the time until the page is shown on the screen. However, this metric requires recording and manually analyzing page load videos, a cumbersome and non-scalable process. Other researchers use `load` [7] or `DOMContentLoaded` [18] events logged in the HTTP

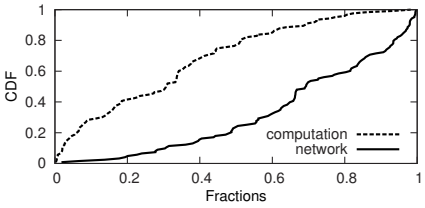


Figure 7: Fractions of time on computation v.s. network on the critical path.

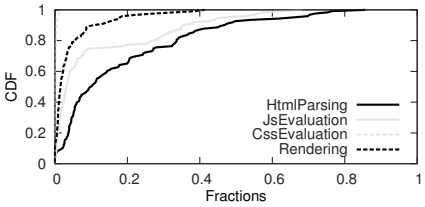


Figure 8: A breakdown of computational time to total page load time on the critical path.

Archival Record (HAR) [12] to indicate the end of the page load process. Since we can tap directly into the browser, we do not need to rely on the external HAR.

We perform additional experiments with varying location, compute power, and Internet speeds. To exclude bias towards popular pages, we also experiment on 200 random home pages that we choose from the top 1 million Alexa websites. We summarize our results of these experiments in §5.5.

5.2 Identifying load bottlenecks (no caching)

The goal of this section is to characterize bottleneck activities that contribute to the page load time. Note that all of these results focus on delays on critical paths. In aggregate, a typical page we analyzed contained 32 objects and 6 activities on the critical path (all median values).

Computation is significant: Figure 7 shows that 35% of page load time in the critical path is spent on computation; therefore computation is a critical factor in modeling or simulating page loads. Related measurements [14] do not estimate this computational component because they treat the browser engine as a black box.

We further break down computation into HTML parsing, JavaScript and CSS evaluation, and rendering in Figure 8. The fractions are with respect to the total page load time. Little time on the critical path is spent on firing timers or listeners, and so we do not show them. Interestingly, we find that HTML parsing costs the most in computation, at 10%. This is likely because: (i) many pages contain a large number of HTML tags, requiring a long time to convert to DOM; (ii) there is a significant amount of overhead when interacting with other components. For example, we find an interval of two milliseconds between reception of the first block of an HTML page and parsing the first HTML tag. JavaScript evaluation is also significant as Web pages embed more and

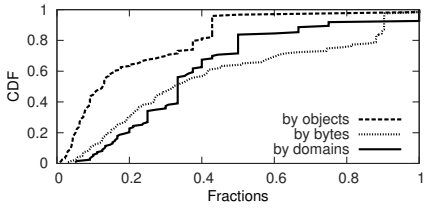


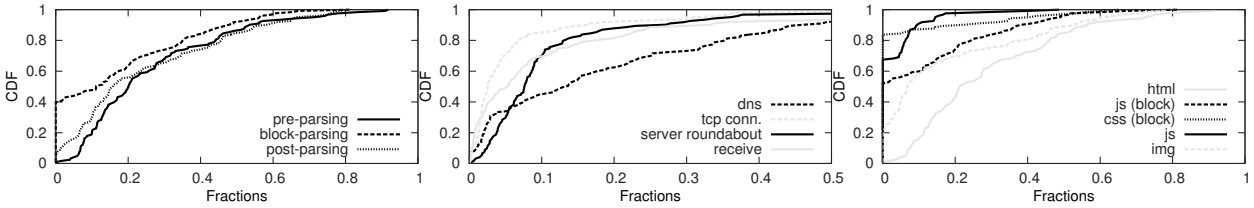
Figure 9: Fractions of domains, objects, and bytes on critical paths.

more scripts. In contrast, CSS evaluation and rendering only cost a small fraction of page load time. This suggests that optimizing CSS is unlikely to be effective at reducing page load time.

Network activities often block parsing. First, we break down network time by how it interacts with HTML parsing in Figure 10(a): pre-parsing, block-parsing, and post-parsing. As before, the fractions are with respect to the total page load time. The pre-parsing phase consists of fetching the first chunk of the page during which no content can be rendered. 15% of page load time is spent in this phase. The post-parsing phase refers to loading objects after HTML parsing (e.g., loading a_{10} and a_{12} in Figure 3). Because rendering can be done before or during post parsing, post parsing is less important, though significant. Surprisingly, much of the network delay in the critical path blocks parsing. Recall that network loads can be done in parallel with parsing unless there are dependencies that block parsing, e.g., JavaScript downloads. Parsing-blocking downloads often occur at an early stage of HTML parsing that blocks loading subsequent embedded objects. The result suggests that a large portion of the critical path delay can be reduced if the pages are created carefully to avoid blocked parsing.

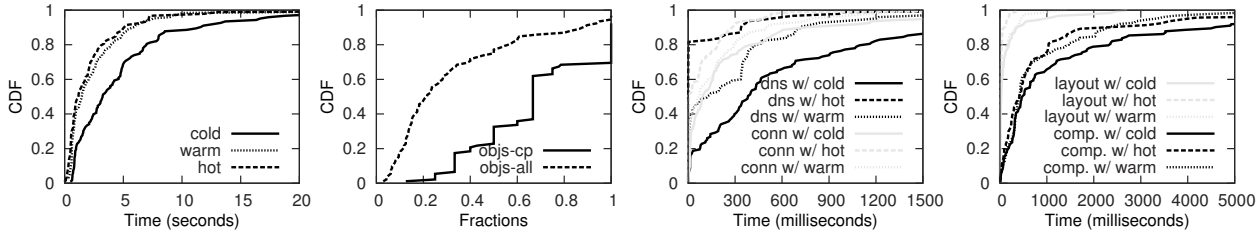
Second, we break down network time by functionality in Figure 10(b): DNS lookup, TCP connection setup, server roundabouts, and receive time. DNS lookup and TCP connection setup are summed up for all the objects that are loaded, if they are on the critical path. Server roundabout refers to the time taken by the server to process the request plus a round trip time; again for every object loaded on the critical path. Finally, the receive time is the total time to receive each object on the critical path. DNS lookup incurs almost 13% of the critical path delay. To exclude bias towards one DNS server, we repeated our experiments with an OpenDNS [21] server, and found that the lookup time was still large. Our results suggest that reducing DNS lookup time alone can reduce page load time significantly. The server roundabout time is 8% of the critical path.

Third, we break down network time by MIME type in Figure 10(c). We find that loading HTML is the largest fraction (20%) and mostly occurs in the pre-parsing phase. Loading images is also a large fraction on critical



(a) By phase. (b) By functionality. (c) By mime type.

Figure 10: A breakdown of fractions of network time on the critical path.



(a) Comparing PLT. (b) Object downloads reduced due to hot load. (c) TCP setup and DNS lookup. (d) Layout and computation time.

Figure 11: Warm and hot loads results. All results are a fraction of total page load time.

paths. Parsing-blocking JavaScript is significant on critical paths while asynchronous JavaScript only contributes a small fraction. Interestingly, we find a small fraction of CSS that blocks JavaScript evaluation and thus blocks HTML parsing. This blocking can be reduced simply by moving the CSS tag after the JavaScript tag. There is almost no non-blocking CSS on critical paths, and therefore we omit it in Figure 10(c).

Last, we look at the page load time for external objects corresponding to Web Analytics and Ads but not CDNs. We find that one fourth of Web pages have downloads of external objects on their critical path. Of those pages, over half spends 15% or more page load time to fetch external objects and one even spends up to 60%.

Most object downloads are non-critical: Figure 9 compares all object downloads and the object downloads only on the critical path. Interestingly, only 30% bytes of content is on critical paths. This suggests that minifying and caching content may not improve page load time, unless they reduce content downloaded along the critical path. We analyze this further in the next section. Note that object downloads off the critical paths are not completely unimportant. Degrading the delay of some activities that are not on the critical path may cause them to become critical.

5.3 Identifying load bottlenecks (with caching)

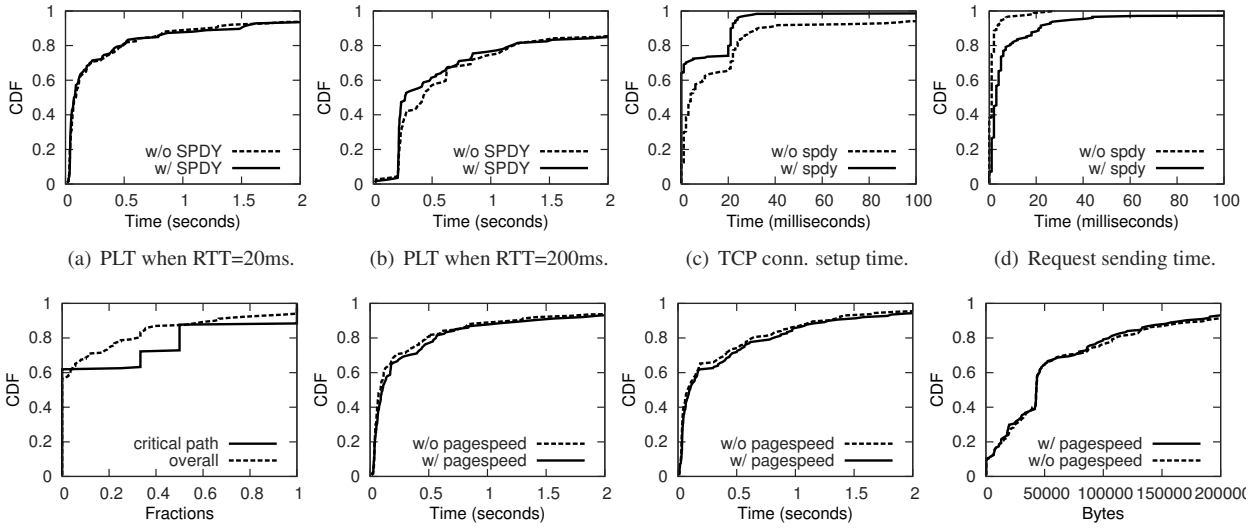
We analyze the effects of caching under two conditions: hot load and warm load. Hot loads occur when a page is loaded immediately after a cold load. Warm loads occur when a page is loaded a small amount of time after a cold load when the immediate cache would have expired. We set a time interval of 12 minutes for warm loads. Both cases are common scenarios, since users often reload pages immediately as well as after a short period of time.

Caching gains are not proportional to saved bytes. Figure 11(a) shows the distributions of page load times under cold, warm, and hot loads. For 50% of the pages, caching decreases page load time by over 40%. However, further analysis shows that 90% of the objects were cached during the experiments. In other words, the decrease in page load time is not proportional to the cached bytes. To understand this further, we analyze the fraction of cached objects that are on and off the critical path, during hot loads. Figure 11(b) shows that while caching reduces 65% of total object loads (marked *objs-all*), it only reduces 20% of object loads on the critical path (marked *objs-cp*). Caching objects that are not on the critical path leads to the disproportional savings.

Caching also reduces computation time. Figures 11(c) and 11(d) compare the network times and computation times of hot, warm, and cold loads, on the critical path. As expected, hot and warm loads reduce DNS lookup time and TCP connection setup. Especially during hot loads, DNS lookup time is an insignificant fraction of the page load time in contrast to cold loads. Interestingly, caching not only improves network time, but also computation time. Figure 11(d) shows the time taken by compute and layout activities during page load time. The figure suggests that modern browsers cache intermediate computation steps, further reducing the page load time during hot and warm loads.

5.4 Evaluating proposed techniques

This section evaluates two Web optimization techniques, SPDY [28] and mod_pagespeed [19].



(a) PLT when RTT=20ms. (b) PLT when RTT=200ms. (c) TCP conn. setup time. (d) Request sending time.

(e) # of TCP connection set ups when using SPDY. (f) PLT for top web pages. (g) PLT for random pages. (h) # bytes on critical path.

Figure 12: SPDY and mod_pagespeed results.

5.4.1 SPDY

SPDY is an application-level protocol in place of HTTP. The key ideas in SPDY are—(i) Multiplexing HTTP transactions into a single TCP connection to reduce TCP connection setup time and to avoid slow start, (ii) prioritizing some object loads over others (e.g., JavaScript over images), and (iii) reducing HTTP header sizes. SPDY also includes two optional techniques—Server Push and Server Hint. Exploiting these additional options require extensive knowledge of Web pages and manual configurations. Therefore, we do not include them here.

We evaluate SPDY with controlled experiments on our own server with 2.4GHz 16 core CPU 16GB memory and Linux kernel 2.6.39. By default, we use a link with a controlled 10Mbps bandwidth, and set the TCP initial window size to 10 (increased from the default 3, as per SPDY recommendation). We use the same set of pages as the real-world experiments and download all embedded objects to our server[1] to avoid domain sharding [20]. We run SPDY version 2 over SSL.

SPDY only improves performance at high RTTs. Figure 12(a) and Figure 12(b) compares the page load time of SPDY versus non-SPDY (i.e., default HTTP) under 20ms and 200ms RTTs, respectively. SPDY provides few benefits to page load time under low RTTs. However, under 200ms RTT, SPDY improves page load time by 5%–40% for 30% of the pages. We conduct additional experiments by varying the TCP initial window size and packet loss, but find that the results are similar to the default setting.

[1]Because we are unable to download objects that are dynamically generated, we respond to these requests with a HTTP 404 error.

SPDY reduces TCP connection setup time but increases request sending time. To understand SPDY performance, we compare SPDY and non-SPDY network activities on the critical path and break down the network activities by functionality; note that SPDY does not affect computation activities. For the 20ms RTT case, Figure 12(c) shows that SPDY significantly reduces TCP connection setup time. However, since SPDY delays sending requests to create a single TCP connection, Figure 12(d) shows that SPDY increases the time taken to send requests. Other network delays such as server roundabout time and total receive time remained similar.

Further, Figure 12(e) shows that although SPDY reduces TCP setup times, the number of TCP connection setups in the critical path is small. Coupled with the increase in request sending time, the total improvement due to SPDY cancels out, resulting in no net improvement in page load time.

The goal of our evaluation is to explain the page load behavior of SPDY using critical path analysis. Improving SPDY's performance and leveraging SPDY's optional techniques are part of future work.

5.4.2 mod_pagespeed

mod_pagespeed [19] is an Apache module that enforces a number of best practices to improve page load performance, by minifying object sizes, merging multiple objects into a single object, and externalizing and/or inlining JavaScripts. We evaluate mod_pagespeed using the setup described in §5.4.1 with a 20ms RTT.

Figure 12(f) compares the page load times with and without mod_pagespeed on top 200 Alexa pages. mod_pagespeed provides little benefits to page load time.

Since the top 200 pages may be optimized, we load 200 random pages from the top one million Alexa Web pages. Figure 12(g) shows that mod_pagespeed helps little even for random pages.

To understand the mod_pagespeed performance, we analyze minification and merging multiple objects. The analysis is based on loading the top 200 Web pages. Figure 12(h) shows that the total number of bytes downloaded on the critical path remains unchanged with and without mod_pagespeed. In other words, minifying object does not reduces the size of the objects loads on the critical path, and therefore does not provide page load benefits. Similarly, our experiments show that merging objects does not reduce the network delay on the critical path, because the object loads are not the bottleneck (not shown here). These results are consistent with our earlier results (Figure 9) that shows that only 30% of the object loads are on the critical path. We were unable to determine how mod_pagespeed decides to inline or externalize JavaScripts, and therefore are unable to conduct experiments on how inlining/externalizing affects page load performance.

5.5 Summarizing results from alternate settings

In addition to our default experiments, we conducted additional experiments: (i) with 2 machines, one with 2GHz dual core 4GB memory, and another with 2.4GHz dual core 2GB memory, (ii) in 2 different locations, one at UMass Amherst with campus Ethernet connectivity, and another in a residential area in Seattle with broadband connectivity, and (iii) using 200 random Web pages chosen from the top 1 million Alexa Web pages. All other parameters remained the same as default.

Below, we summarize the results of our experiments:

- The fraction of computation and network times on the critical path were quantitatively similar in different locations.
- The computation time as a fraction of the total page load time increased when using slower machines. For example, computation time was 40% of the critical path for the 2GHz machine, compared to 35% for the 3GHz machine.
- The 200 random Web pages experienced 2x page load time compared to the top Web pages. Of all the network components, the server roundabout time of random pages (see Figure 10(b)) was 2.3x of that of top pages. However, the total computation time on the critical path was 12% lower compared to the popular pages, because random pages embed less JavaScript.

6 Discussion

In this work, we have demonstrated that WProf can help identify page load bottlenecks and that it can help evaluate evolving techniques that optimize page loads. WProf can be potentially used in several other applications. Here, we briefly discuss two applications: (i) Conducting what-if analysis, and (ii) identifying potential Web optimizations.

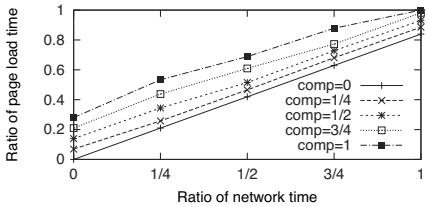


Figure 13: Median reduction in page load times if computation and network speeds are improved.

What-if analysis: As CPUs and networks evolve, the question we ask is—how can page load times benefit from improving the network speed or CPU speed? We use the detailed dependency graphs generated by WProf to conduct this what-if analysis. Figure 13 plots the reduction in page load time for a range of <network speedup, CPU speedup> combinations. When computational time is zeroed but the network time is unchanged, the page load time is reduced by 20%. If the network time is reduced to one fourth, but the computational time is unchanged, 45% of the page load time is reduced. Our results suggest that speeding up both network and computation together is more beneficial than just improving one of them. Note that our what-if analysis is limited as it does not capture all lower-level dependencies and constraints, e.g., TCP window sizes. We view our results here as estimates and leave a more extensive analysis for future work.

Potential optimizations: Our own experience and studies with WProf suggest a few optimization opportunities. We find that synchronous JavaScript significantly affects page load time, not only because of loading and evaluation, but also because of block-parsing (Figure 10(a)). Asynchronous JavaScript or in-lining the scripts can help reduce page load times. To validate this opportunity, we manually transform synchronous JavaScript to `async` on top five pages and find that this helps page load time. However, because asynchronous JavaScript may alter Web pages, we need further research to understand how and when JavaScript transformation affects Web pages. Another opportunity is with respect to prioritizing object loading according to the dependency graph. Prioritization can be either implemented at the application level such as SPDY or reflected using latency-reducing techniques [31]. For example, prioritizing JavaScript loads can decrease the time that HTML parsing is blocked. Recently, SPDY considers applying the dependency graph to prioritize object loading in version 4 [29]. Other op-

portunities include parallelizing page load activities and more aggressive preloading strategies, both of which require future exploration.

7 Related Work

Profiling page load performance: Google Pagespeed Insight [22] includes a (non open-source) critical path explorer. The explorer presents the critical path for the specific page load instance, but does not extract all dependencies inherent in a page. For example, the explorer does not include dependencies due to resource constraints and eager/late binding. Also, the explorer will likely miss dependencies if objects are cached, if the network speed improves, if a new CDN is introduced, if the DNS cache changes, and many other instances. Therefore, it is difficult to conduct what-if-analysis or to explain the behavior of Web page optimizations using the explorer.

The closest research to WProf is WebProphet [16] that identifies dependencies in page loads by systematically delaying network loads, a technique we borrow in this work. The focus of WebProphet is to uncover dependencies only related to object loading. As a result,, WebProphet does not uncover dependencies with respect to parsing, rendering, and evaluation. WebProphet also predicts page load performance based on its dependency model; we believe that WProf's more complete dependency model can further improve the page load performance prediction.

Tools such as Firebug [11], Developer Tools [8], and more recently HAR [12] provide detailed timings information about object loading. While their information is important to augment WProf profiling, they do not extract dependencies and only focus on the networking aspect of page loads.

Web performance measurements: Related measurement studies focus on either network aspects of Web performance or macro-level Web page characteristics. Ihm and Pai [14], presented a longitudinal study on Web traffic, Ager et al. [2] studied the Web performance with respect to CDNs, and Huang et al. [13] studied the page load performance under a slower cellular network. Butkiewicz et al. [7] conducted a macro-level study, measuring the number of domains, number of objects, JavaScript, CSS, etc. in top pages. Others [26, 15] studied the effect of dynamic content on performance. Instead, WProf identifies internal dependencies of Web pages that let us pinpoint bottlenecks on critical paths.

Improving page load performance: There has been several efforts to improve page load performance by modifying the page, by making objects load faster, or by reducing computational time. At the Web page level, mod_pagespeed [19] and Silo [18] modify the structure and content of the Web page to improve page load. At the networking level, SPDY [28], DNS pre-resolution [9], TCP pre-connect [30], TCP fast open [24], ASAP [35] and other caching techniques [34, 32] reduce the time taken to load objects. At the computation level, inline JavaScript caching [18] and caching partial layouts [18, 17] have been proposed. While these techniques provide improvement for certain aspects of page loads, the total page load performance depends on several inter-related factors. WProf helps understand these factors to guide effective optimization techniques.

8 Conclusion

In this paper, we abstract a model of browser dependencies, and design WProf, a lightweight, in-browser profiler that extracts dependency graphs of any given page. The goal of WProf is to identify bottleneck activities that contribute to the page load time. By extensively loading hundreds of pages and performing critical path analysis on their dependency graphs, we find that computation is a significant factor and makes up as much as 35% of the page load time on the critical path. We also find that synchronous JavaScript evaluation plays a significant role in page load time because it blocks parsing. While caching reduces the size of downloads significantly, it is less effective in reducing page load time because loading does not always affect the critical path. We conducted experiments over SPDY and mod_pagespeed. While the effects of SPDY and mod_pagespeed vary over pages, surprisingly, we find that they help very little on average. In the future, we plan to extend our dependency graphs with more lower-level dependencies (e.g., in servers) to understand how page loads would be affected by these dependencies.

Acknowledgements

We thank our shepherd, Rebecca Isaacs, and the anonymous reviewers for their feedback. We thank Xiaozheng Tie for setting up machines at UMass.

References

[1] Above the fold time. `http://www.webperformancetoday.com/`.

[2] B. Ager, W. Muhlbauer, G. Smaragdakis, and S. Uhlig. Web Content Cartography. In *Proc. of the SIGCOMM conference on Internet Measurement Conference (IMC), 2011*.

[3] Alexa - The Web Information Company. `http://www.alexa.com/topsites/countries/US`.

[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] 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*.

[6] National Broadband Map. http://www.broadbandmap.gov/.

[7] 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*.

[8] Chrome Developer Tools. https://developers.google.com/chrome-developer-tools/docs/overview.

[9] DNS pre-resolution. http://blogs.msdn.com/b/ie/archive/2011/03/17/internet-explorer-9-network-performance-improvements.aspx.

[10] Dummynet. http://info.iet.unipi.it/~luigi/dummynet/.

[11] Firebug. http://getfirebug.com/.

[12] HTTP Archive. http://httparchive.org/.

[13] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing application performance differences on smartphones. In *Proc. of the international conference on Mobile systems, applications, and services (Mobisys), 2010*.

[14] S. Ihm and V. S. Pai. Towards understanding modern web traffic. In *Proc. of the SIGCOMM conference on Internet Measurement Conference (IMC), 2011*.

[15] E. Kiciman and B. Livishits. AjaxScope: A Platform for Remotely Monitoring the Client-side Behavior of Web 2.0 Applications. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP), 2007*.

[16] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang. WebProphet: automating performance prediction for web services. In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI), 2010*.

[17] L. A. Meyerovich and R. Bodik. Fast and parallel webpage layout. In *Proc. of the international conference on World Wide Web (WWW), 2010*.

[18] J. Mickens. Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads. In *Proc. of USENIX conference on Web Application Development (WebApps), 2010*.

[19] mod_pagespeed. http://www.modpagespeed.com/.

[20] Not as SPDY as you thought. http://www.guypo.com/technical/not-as-spdy-as-you-thought/.

[21] Open DNS. http://www.opendns.com/.

[22] Google Pagespeed Insights. https://developers.google.com/speed/pagespeed/insights.

[23] Shopzilla: faster page load time = 12% revenue increase. http://www.strangeloopnetworks.com/resources/infographics/web-performance-and-ecommerce/shopzilla-faster-pages-12-revenue-increase/.

[24] 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*.

[25] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, USA, 1989.

[26] F. Schneider, S. Agarwal, T. Alpcan, and A. Feldmann. The New Web: Characterizing AJAX Traffic. In *Proc. of Passive and Active Measurement Conference (PAM), 2008*.

[27] Selenium. http://seleniumhq.org/.

[28] SPDY. http://dev.chromium.org/spdy.

[29] Proposal for Stream Dependencies in SPDY. https://docs.google.com/document/d/1pNj2op5Y4r1AdnsG8bapS79b11iWDCStjCNHo3AWD0g/edit.

[30] TCP pre-connect. http://www.igvita.com/2012/06/04/chrome-networking-dns-prefetch-and-tcp-preconnect/.

[31] A. Vulimiri, O. Michel, P. B. Godfrey, and S. Shenker. More is Less: Reducing Latency via Redundancy. In *Proc. of the ACM Workshop on Hot Topics in Networks (HotNets-XI), 2012*.

[32] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How far can client-only solutions go for mobile browser speed? In *Proc. of the international conference on World Wide Web (WWW), 2012*.

[33] The Webkit Open Source Project. http://www.webkit.org/.

[34] K. Zhang, L. Wang, A. Pan, and B. B. Zhu. Smart caching for web browsers. In *Proc. of the international conference on World Wide Web (WWW), 2010*.

[35] W. Zhou, Q. Li, M. Caesar, and B. Godfrey. ASAP: A Low-Latency Transport Layer. In *Proc. of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2011*.

Dasu: Pushing Experiments to the Internet's Edge

Mario A. Sánchez* John S. Otto* Zachary S. Bischof* David R. Choffnes†
Fabián E. Bustamante* Balachander Krishnamurthy‡ Walter Willinger‡
*Northwestern U. †U. of Washington ‡AT&T Labs-Research

Abstract

We present Dasu, a measurement experimentation platform for the Internet's edge. Dasu supports both controlled network experimentation and broadband characterization, building on public interest on the latter to gain the adoption necessary for the former. We discuss some of the challenges we faced building a platform for the Internet's edge, describe our current design and implementation, and illustrate the unique perspective it brings to Internet measurement. Dasu has been publicly available since July 2010 and has been installed by over 90,000 users with a heterogeneous set of connections spreading across 1,802 networks and 147 countries.

1 Introduction

Our poor visibility into the network hampers progress in a number of important research areas, from network troubleshooting to Internet topology and performance mapping. This well-known problem [8, 39] has served as motivation for several efforts to build new testbeds or expand existing ones by recruiting increasingly large and diverse sets of measurement vantage points [15, 33, 37]. However, capturing the diversity of the commercial Internet (including, for instance, end-hosts in homes and small businesses) at sufficient scale remains an elusive goal [17, 25].

We argue that, at its roots, the problem is one of incentives. Today's measurement and experimentation platforms offer two basic incentive models for adoption – cooperative and altruistic. In cooperative platforms such as PlanetLab [31] and RIPE Atlas [35] an experimenter interested in using the system must first become part of it. Other platforms such as SatelliteLab [15] and DIMES [37] have opted instead for an altruistic approach in which users join the platform for the betterment of science. All these efforts build on the assumption, sometimes implicit, that the goals of those hosting the platform and the experimenters that use it are aligned. As much of the Internet's recent growth occurs in residential broadband networks [1] this assumption no longer holds.

This paper presents Dasu — a platform for network measurement experimentation and the Internet's edge built with an alternate model that explicitly aligns the objectives of platform hosts and experimenters. Dasu[1] is designed to support both broadband characterization and Internet measurement experiments and leverage their synergies. Both functionalities benefit from wide network coverage to capture network and broadband service diversity. Both can leverage continuous availability to capture time-varying changes in broadband service levels and to enable long-running and time-dependent measurement experiments. Both must support dynamic extensibility to remain effective in the face of ISP policy changes and to enable purposefully-designed, controlled Internet experiments. Finally, both functionalities must be available at the edge of the network to capture the end users' view of the provided services and offer visibility into this missing part of the Internet [9].

This paper focuses on Dasu's support for Internet measurement experimentation, outlining our current design and how it addresses some of the key challenges raised by our goals.[2] Dasu has been publicly available since June 2010 and is currently in use by 90,222 users with a heterogeneous set of connections spreading over 1,802 networks and across 147 countries.

We make the following contributions in this work:

- We present the design and implementation of Dasu — an extensible platform for measurement experimentation from the Internet's edge.
- We describe the current deployment of Dasu and present results demonstrating how the participating nodes collectively offer broad network coverage,

[1] Dasu is a Japanese word that can mean "to reveal" or "to expose".
[2] Please see [6] for a general description of Dasu's support for broadband characterization.

high availability and fine-grained synchronization to enable Internet measurement experimentation.

- We use three case studies to illustrate the unique perspective that a platform like Dasu brings to Internet measurement. In the process, we demonstrate Dasu's capabilities to (i) simplify traditional measurements (e.g., examining routing asymmetry), (ii) reveal fundamental shortcomings in existing measurement efforts (e.g., mapping AS-level topology), and (iii) conduct novel experiments for original system evaluations (e.g., examining the effectiveness of a recently-proposed DNS extension).

The rest of this paper is structured as follows. We put our work in context and provide further motivation in Sec. 2. In Sec. 3 and 4 we outline the design and implementation of Dasu and characterize our current deployment. We present cases studies that illustrate the benefits of a measurement experimentation platform that runs at the Internet's edge in Sec. 5. Finally, we discuss future work and present our conclusions in Sec. 6.

2 Background and Motivation

The lack of network and geographic diversity in current Internet experimentation platforms is well a known problem [8, 39]. Most Internet measurement and system evaluation studies rely on dedicated infrastructures [3, 5, 7, 31] which provide relatively continuous availability at the cost of limited vantage point diversity (i.e. with nodes primarily located in well-provisioned academic or research networks that are not representative of the larger Internet).

Several research projects have pointed out the pitfalls when attempting to generalize results of network measurements taken with a limited network perspective (e.g. [8, 10, 27, 32, 45]). For example, consider the differences in paths between PlanetLab nodes and between nodes in residential networks. These two sets traverse different parts of the network [9], exhibit different latency and packet loss characteristics [12, 20] and result in different network protocol behaviors [16].

2.1 Goals and Approach

An experimental platform for the Internet should be deployed at scale to capture network and service diversity. It should be hosted at the network edge to provide visibility into this opaque part of the Internet. Such a platform should allow dynamic extensibility in order to enable purposefully-designed, controlled measurement experiments, without compromising end-host security. To support time-dependent and long-running experiments, it should offer (nearly) continuous availability. Last, it should facilitate the design and deployment of experiments at the network edge while controlling the impact on the resources of participating nodes and the underlying network resources.

Dasu is an experimental platform designed to match these goals. To capture the diversity of the commercial Internet, Dasu supports both Internet measurement experimentation and broadband characterization and leverages their synergies. In its current version, Dasu is built as an extension to the most popular large-scale peer-to-peer system – BitTorrent.[3] The typical usage patterns and comparatively long session times of BitTorrent users means that Dasu can attain nearly continuous availability to launch measurement experiments. More importantly, by leveraging BitTorrent's popularity, Dasu attains the necessary scale and coverage at the edge of the network. Dasu is tailored for Internet network experimentation and, unlike general-purpose Internet testbeds such as PlanetLab, does not support the deployment of planetary-scale network services.

2.2 Challenges

Both strengths and challenges of a platform like Dasu stem from its inclusion of participating nodes at the Internet's edge. For one, the increased network coverage from these hosts comes at the cost of higher volatility and leaves the platform at the "mercy" of end users' behavior. The types of experiments possible in such a platform depend thus on the clients' availability and session times since these partially determine the maximum length of the experiment that can be safely assigned to clients. Such a platform must provide a scalable way to share measurement resources among concurrent experiments with a dynamic set of vantage points. It must also guarantee the safety of the volunteer nodes where it is hosted (for instance, by restricting the execution environment), and ensure secure communication with infrastructure servers. Last, to control the impact that experiments may have on underlying network and system resources, the system must support coordinated measurements among large numbers of hosts worldwide, each of which is subject to user interaction and interference.

2.3 Related Work

Dasu shares goals with and builds upon ideas from several prior large-scale platforms targeting Internet experimentation. Most active measurement and experimentation research relies on dedicated infrastructures (PlanetLab [31], Ark [7], Looking Glass servers). Such infrastructures provide relatively continuous availability and nearly continuous monitoring at the cost of limited vantage point diversity. Dasu targets the increasingly "invisible" portions of the Internet, relying on a direct

[3] A stand-alone version of Dasu has been developed and we plan to release it in June 2013.

incentive model to ensure large-scale adoption at the Internet edge.

Several related projects use passive measurements or restricted active measurements from volunteer platforms to capture this same perspective (e.g., [15, 33, 35, 37, 38, 42]). In contrast, Dasu is a software-based solution with a much broader set of measurement vantage points that has been achieved by altruistic and hardware-based systems, and supports a programmable interface that enables complex, coordinated measurements across the participating hosts. As such, Dasu shares some design goals with Scriptroute [40] and SatelliteLab [15]). Unlike Scriptroute, Dasu is intended for large scale deployment on end users' machines, and relies on incentives for user adoption at scale. Dasu also enables programable measurements without requiring root access, avoiding potential security risks and barriers to adoption. SatelliteLab adopts an interesting two-tier architecture that links end hosts (satellites) to PlanetLab nodes and separates traffic forwarding (done by satellites) from code execution. In Dasu, experiment code generates traffic directly from hosts at the network edge.

Several systems have proposed leveraging clients in a P2P system to measure, diagnose and predict the performance of end-to-end paths (e.g., [11, 28]. Dasu moves beyond these efforts, exploring the challenges and opportunities in supporting programmable experimentation from volunteer end hosts.

3 Dasu Design

In this section, we provide an overview of Dasu's design, discuss several system's components and briefly describe the API supporting measurement experiments.

3.1 System Overview

Dasu is composed of a distributed collection of clients and a set of management services. Dasu clients provide the desired coverage and carry on the measurements needed for broadband characterization and Internet experimentation. The *Management Services*, comprising the Configuration, Experiment Administration, Coordination and Data services, distribute client configuration and experiments and manage data collection. Figure 1 presents the different components and their interactions.

Upon initialization, clients use the *Configuration Service* to announce themselves and obtain various configuration settings including the frequency and duration of measurements as well as the location to which experiment results should be reported. Dasu clients periodically contact the Experiment Administration Service, which assigns measurement tasks, and the Coordination Service to submit updates about completed probes and retrieve measurement limits for the different experiment tasks. Finally, clients use the Data Service to report

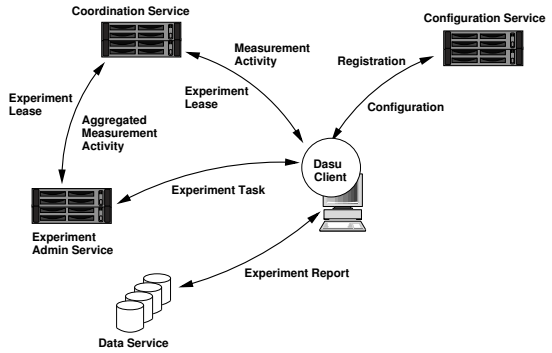


Figure 1: *Dasu system components.*

the results of completed experiments as they become available.

3.2 Experiment Specification

Dasu is a dynamically extensible platform designed to facilitate Internet measurement experimentation while controlling the impact on hosts' resources and the underlying network. A key challenge in this context is selecting a programming interface that is both flexible (i.e., supports a wide range of experiments) and safe (i.e., does not permit run-away programs). We rejected several approaches based on these constraints and our platforms goals. These include offering only a small and fixed set of measurement primitives as they would limit flexibility. We also avoided providing arbitrary binary execution as handling the ramifications of such an approach would be needlessly complex.

We opted for a rule-based declarative model for experiment specification in Dasu. In this model, a rule is a simple `when-then` construct that specifies the set of actions to execute when certain activation conditions hold. A rule's left-hand side is the conditional part (`when`) and states the conditions to be matched. The right-hand side is the consequence or action part of the rule (`then`) i.e., the list of actions to be executed. Condition and action statements are specified in terms of read/write operations on a shared working memory and invocation of accessor methods and measurement primitives. A collection of rules form a program and a set of related programs define an experiment.

The rule-based model provides a clean separation between experiment logic and state. In our experience, this has proven to be a flexible and lightweight approach for specifying and controlling experiments. Experiment logic is centralized, making it easy to maintain and extend. Also, strict constraints can be imposed on rule syntax, enabling safety verification through simple static program analysis.

Dasu provides an extensible set of measurement primitives (modules) and a programmable API to combine them into measurement experiments. Tables 1

Method	Params.	Description
addProbeTask	<probe> <params> [<times>] [<when>]	Submit measurement request of the specified type.
commitResult	<report>	Submit completed experiment results to data server.
getClientIPs	[]	Return network information about the client including the list of IP addresses assigned (both public and private).
getDnsServers	[]	Return the list of DNS servers configured at the client.
getEnvInfo	[]	Return information about the plugin and the host node, including OS information and types of measurement probes available to the experimenter.

Table 1: *Dasu API – Methods.*

Probe	Params.	Description
PING	<dest-list (IP/name)>	Use the local host ping implementation to send $ECHO\_REQUEST$ packets to a host.
TRACEROUTE	<dest-list (IP/name)>	Print the route packets take to a network host.
NDT	[<server>]	Run the M-Lab Network Diagnostic Tool [29].
DNS	[<server>] | [<timeout>] | [<tcp/udp>] | [<options>] | <DNS-msg>] | <dest-list>	Submit DNS resolution request to a set of servers.
HTTP	[server] | [<port>] | [<HTTP-Req>] | <url-list>	Submit HTTP request to a a given $< host, port >$ pair.

Table 2: *Dasu API – Measurement modules currently supported.*

and 2 provide a summary of this API and the current set of measurement primitives supported. The API includes some basic accessor methods (e.g. `getClientIps`, `getDnsServers` and `getEnvInfo`). The method `addProbeTask` serves to request the execution of measurements at a given point in time. The `commitResult` method allows results from the experiment to be submitted to the Data Service after completion.

Dasu provides low-level measurement tools that can be combined to build a wide range of measurement experiments. Currently available measurement primitives include traceroute, ping, Network Diagnostic Tool (NDT) [29], HTTP GET and DNS resolution. While this set is easily extensible (by the platform administrators) we have found it sufficient to allow complex experiments to be specified clearly and concisely. For instance, the experiment for the Routing Asymmetry case study (Sec. 5.1) was specified using only 3 different rules with an average of 24 lines of code per rule.

Measurements primitives are invoked asynchronously by the *Coordinator*, which multiplexes resources across experiments. Progress and results are communicated through a shared *Working Memory*; through this working memory, an experiment can also chain rules that schedule measurements and handle results.

In addition to these active measurements, Dasu leverages the naturally-generated BitTorrent traffic as passive measurements (particularly in the context of broadband characterization [6]) by continuously monitoring the end-host Internet connection. Devising an interface to expose these passively collected measurements to experimenters is part of future work.

A Simple Example. To illustrate the application of rules, we walk through the execution of a simple experiment for debugging high latency DNS queries. Figure 2 lists the rules that implement this experiment. When rule #1 is triggered, it requests a DNS resolution for a domain name using the client's configured DNS server. When the DNS lookup completes, rule #2 extracts the IP address from the DNS result and schedules a ping measurement. After the ping completes, rule #3 checks the ping latency to the IP address and schedules a traceroute measurement if this is larger than 50 ms.

3.3 Delegating Code Execution to Clients

Dasu manages concurrent experiments, including resource allocation, via the Experiment Administration Service. As clients become available, they announce their specific characteristics (such as client IP prefix, connection type, geographic location and operating system) and request new experiment tasks. The *Experiment Administration (EA) Service* assigns tasks to a given client based on experiment requirements and characteristics of available clients (e.g. random sample of DSL users in Boston).

In the simplest of experiments, every Dasu client assigned to an experiment will receive and execute the same experiment task (specified as a stand-alone rules file). Dasu also enables more sophisticated experiments where experimenters specify which clients to use and how to execute tasks based on client characteristics.

```
rule "(1) Resolve IP address through local DNS"
 when
  $fact : FactFireAction(action=="resolveIp");
 then
  addProbeTask(ProbeType.DNS, "example.com");
end

rule "(2) Handle DNS lookup result"
 when
  $dnsResult : FactDnsResult(toLookup=="example.com")
 then
  String ip = $dnsResult.getSimpleResponse();
  addProbeTask(ProbeType.PING, ip);
end

rule "(3) Handle ping measurement result"
 when
  $pingResult : FactPingResult()
 then
  if ( $pingResult.getRtt() > 50 )
   addProbeTask(ProbeType.TRACEROUTE, $pingResult.ip );
end
```

Figure 2: *Example measurement experiment for debugging high latency DNS queries.*

Dasu adopts a two-tiered architecture for the EA Service, with a primary server, responsible for resource allocation, and a number of secondary servers in charge of particular experiments. The *Primary EA* server acts as a broker, allocating clients to experiments, by assigning them to the responsible secondary server, based on clients' characteristics and resource availability. The *Secondary EA server* is responsible for task parameterization and allocation of tasks to clients according to the experiment's logic. While the customized task assigned to a client is generated by the experiment's secondary server, all communication with Dasu clients is mediated by the primary server who is responsible for authenticating and digitally signing the assigned experiments.

Submitting External Experiments. Dasu supports third-party experiments through the two-tier architecture described above. Authorized research groups host their own Secondary EA server, with security and accountability provided through the Primary EA server.

In addition to providing a safe environment for executing experiments, all experiments submitted to Dasu are first curated and approved by the system administrators before deployment. This curation process serves as another safety check and ensures that admitted experiments are aligned with the platform's stated goals.

3.4 Security and Safety

Safely conducting measurements is a critical requirement for any measurement platform and particularly for one deployed at the Internet edge. We focus on two security concerns: protecting the host and the network when executing experiments. We expand on the former here and discuss the latter in the following section.

To protect the host, Dasu uses a sandboxed environment for safe execution of external code, ensures secure communication with infrastructure servers, and carefully limits resource consumption.

Experiment Sandbox. To ensure the execution safety of external experiments, Dasu confines each experiment to a separate virtual machine, instantiated with limited resources and with a security manager that implements restrictive security polices akin to those applied to unsigned Java applets. In addition, all Dasu experiments are specified as a set of rules that are parsed for unsafe imports at load time, restricting the libraries that can be imported. Dasu inspects the experiment's syntax tree to ensure that only specifically allowed functionality is included and rejects a submitted experiment otherwise.

Secure communication. To ensure secure communication between participating hosts and infrastructure servers, all configuration and experiment rule files served by the EA Service are digitally signed for authenticity and all ongoing communications with the servers (e.g. for reporting results) are established over secure channels.

Limits on resource consumption. Dasu must carefully control the load its experiments impose on the local host, as well as minimize the impact that users' interactions (i.e., with the host and the application) can have on experiments' results. To this end, Dasu limits consumption of hosts' resources[4] and restricts the launching of experiments to periods of low resource utilization; the monitored resources include CPU time, network bandwidth, memory and disk space.

To control CPU utilization, Dasu monitors the fraction of CPU time consumed by each system component (including the base system and each different probe module). Dasu regulates average CPU utilization by imposing time-delays on the activity of individual probe modules whenever their "fair share" of CPU time has been exceeded over the previous monitoring period. Dasu also employs watchdog timers to control for long-running experiments.

To control bandwidth consumption, Dasu passively monitors the system bandwidth usage and launches active measurements only when utilization is below certain threshold (we evaluate the impact of this policy on experiment execution time in Sec. 4.3). Dasu uses the 95th percentile of client's throughput rates measured by NDT to estimate the maximum bandwidth capacity of the host and continuosly monitors host network activity (using the commonly available `netstat` tool). Based on pre-computed estimates of approximate bandwidth consumption for each probe, Dasu limits probe execution by only launching those that will not exceed the predetermined average bandwidth utilization limit. Additionally Dasu relies on a set of predefined limits on the number

[4]Currently 15% of any monitored resource.

of measurement probes of each type that can be launched per monitored interval. While clients are allowed to dispense with their entire budget at once, the combined bandwidth consumed by all probe modules must remain below the specified limit.

To restrict memory consumption, Dasu monitors the allocated memory used by its different data structures and limits, for instance, the number of queued probe-requests and results. Measurement results are offloaded to disk until they can successfully be reported to the Data Service. Disk space utilization is also controlled by limiting the size of the different probe-result logs; older results are dropped first when the pre-determined quota limits have been reached.

3.5 Coordination

In addition to controlling the load on and guaranteeing the safety of volunteer hosts, Dasu must control the impact that measurement experiments collectively may have on the underlying network and system resources. For instance, although the individual launch rate of `ping` measurements is limited, a large number of clients probing the same destination can overload it.

To this end, Dasu introduces two new constructs - *experiment leases* and *elastic budgets*, to efficiently allow the scalable and effective coordination of measurements among potentially thousands of hosts. In the following paragraphs, we describe both constructs and Dasu's approach to coordination.

Experiment Leases. To support the necessary fine-grained control of resource usage, we introduce the concept of experiment leases. In general, a *lease* is a contract that gives its holder specified rights over a set of resources for a limited period of time [19]. An *experiment lease* grants to its holder the right to launch a number of measurement probes, using the common infrastructure, from/toward a particular network location. Origin and/or targets for the probes can be specified as IP-prefixes or domain names (other forms, such as geographic location, could be easily incorporated).

Experiment leases are managed by the EA Service. The Primary EA server ensures that the aggregated use of resources by the different experiments is within the specified bounds. Secondary EA servers are responsible for managing experiment leases to control the load imposed by their particular experiments. To coordinate the use of resources by the Dasu clients taking part in an experiment, we rely on a distributed coordination service [23]. The Coordination Service runs on well-provisioned servers (PlanetLab nodes) using replication for availability and performance. Clients receive the list of coordination servers as part of the experiment description.

Before beginning an experiment, clients must contact a coordinator server to announce they are joining the experiment and obtain an associated lease. As probes are launched, the clients submit periodic updates to the coordination servers about the destinations being probed. The EA Service uses this information to compute estimated aggregate load per destination and to update the associated entries in the experiment lease. Before running a measurement, the Coordinator checks whether it violates the constraint on the number of probes allowed for the associated source and destination, and if so delays it. After a lease expires, the host must request a new lease or extend the previous one before issuing a new measurement. The choice of the lease term presents a trade-off between minimizing overhead on the EA Service versus minimizing client overhead and maximizing its use.

Elastic Budget. An experiment lease grants to its holder the right to launch a number of measurement probes (i.e., *a budget*) from/toward a particular network location. Due to churn and user-generated actions, the number of measurement probes a Dasu client can launch before lease expiration (i.e., the fraction of the allocated budget actually used) can vary widely. To account for this, Dasu introduces the idea of *elastic budgets* that expand and contract based on system dynamics.

Elastic budgets are computed by the EA Service and used to update bounds on experiment leases distributed to Dasu clients. The EA Service calculates the elastic budget periodically based on the current number of clients participating in the experiment, the number of measurement probes allowed, assigned and completed by each client. The EA Service uses this elastic budget to compute measurement probe budgets for the next lease period for each participating client. This approach is well suited for experiments where the server knows a priori what destinations each client should probe. In the case of experiments where the destinations to be probed are not assigned by the server, but obtained by the clients themselves (through a DNS resolution for example), the same approach can be used if we conservatively assume that a client will launch the maximum number of probes per unit of time whenever it is online.

3.6 Synchronization

Dasu also provides support for Internet experiments that require synchronized client operation (e.g. [34, 41]).

For coarse-level synchronization, Dasu clients include a cron-like probe-scheduler that allows the scheduling of measurements for future execution. All Dasu clients periodically synchronize their clocks using NTP. Assuming clients' clocks are closely synchronized, an experiment can request the "simultaneous" launch of measurements

Region	Penetration	Dasu Total	Dasu Total Countries
North America	78.6 %	21.45 %	60 %
Oceania/Australia	67.5 %	3.82 %	6 %
Europe	61.3 %	59.25 %	73 %
L. America/Carib.	39.5 %	1.68 %	65 %
Middle East	35.6 %	1.52 %	73 %
Asia	26.2 %	2.59 %	57 %
Africa	13.5 %	9.66 %	34 %

Table 3: *Internet penetration*[6] *and Dasu coverage (as percentage of its total population of 90,222) by January 2013.*

by a set of clients. We have found this to be sufficient to achieve task synchronization on the order of 1-3 seconds.

For finer-grained synchronization (on the order of milliseconds), Dasu adopts a remote triggered execution model. All synchronized clients must establish persistent TCP connections with one of the coordination servers. These connections are later used to trigger clients actions at a precise moment, taking into account network delays between clients and coordination servers.

4 Deployment

We have implemented Dasu as an extension to a popular BitTorrent client [43] as it offers a large and widespread client population and a powerful interface for extensions. We have made Dasu publicly available since June 2010.

To participating users, Dasu provides information about the service they receive from their ISP [6, 36]. Access to such information has proven sufficient incentive for widespread subscription with over 90K users who have adopted our extension with minimum advertisement.[5]

This section demonstrates how Dasu clients collectively provide broad network coverage, sufficiently high availability and fine-grained synchronization for Internet experimentation.

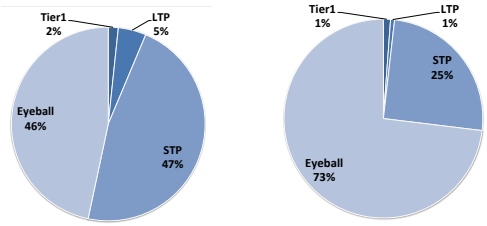
4.1 Dasu Coverage

We show the coverage of Dasu's current deployment in terms of geography and network topology. Table 3 lists broadband penetration in each primary geographic region and compares these numbers with those from our current Dasu's deployment.

Given the high Internet penetration numbers in Europe and North America, the distribution of Dasu clients per region is not surprising. Note, however, the penetration of Dasu clients per region, measured as the percentage of countries covered. As the table shows, Dasu penetration is over 57% for most regions and is particularly high for Latin America/Caribbean (65%) and the Middle East

[5]Upon download, users are informed of both roles of Dasu. Users can, at any point, opt to disable experiments from running and/or reporting performance information, without losing access to Dasu's broadband benchmarking information.

[6]`http://www.internetworldstats.com`



(a) Peer Distribution **(b)** AS Distribution

Figure 3: *Distribution of Dasu peers per AS (left). Distribution of ASes covered by Dasu peers (right).*

(73%), two of the fastest growing Internet regions. Even in Africa Dasu penetration reaches 34%.

We also analyze Dasu's network coverage in terms of ASes where hosts are located. With our existing user-base at the end of January 2013, we have Dasu clients in 1,802 different ASes. We classify these ASes following a recently proposed approach [13], as follows:

- Tier-1: 11 known Tier-1s
- LTP: Large (non tier-1) transit providers and large (global) communications service providers
- STP: Small transit providers and small (regional) communication service providers
- Eyeball: Enterprise customers or access/hosting providers

Figure 3a uses this classification to illustrate where Dasu peers are deployed. As the figure shows, 93% of Dasu peers are located in small transit providers and eyeball ASes; with only minimal presence in large transit and Tier-1 providers. Figure 3b presents the distribution of all the ASes covered by Dasu peers. This figure shows that 73% of the ASes covered by Dasu are eyeball ASes, highlighting the effectiveness of Dasu as a platform for capturing the view from the network edge.

4.2 Dasu Dynamics

In this section, we show that the churn from Dasu clients is sufficiently low to support meaningful experimentation. This churn is a result of both the volatility of Dasu's current hosting application (i.e. BitTorrent) and that of the end systems themselves. In the following analysis, we focus on the hosting application dynamics. In particular, we investigate what portion of clients are online at any moment, and whether their session times support common measurement durations.

First, we analyze Dasu clients' availability, using the percentage of clients online at any given hour over a 31-day period. Figure 4 plots this for the month of January 2013. The fraction of available clients during the period varies, on average, between 39% and 44% of the total number of unique users seen during a day, with a total of 1,473 active unique users for the month. With respect to

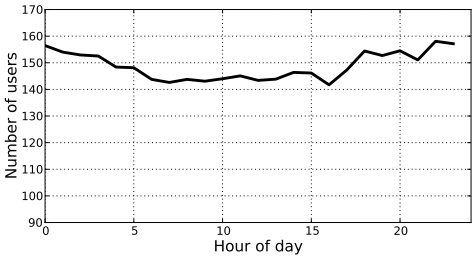


Figure 4: *Number of online Dasu clients over a 24-hour period. The fraction ranges from 39-44% of the total number of unique users, on average.*

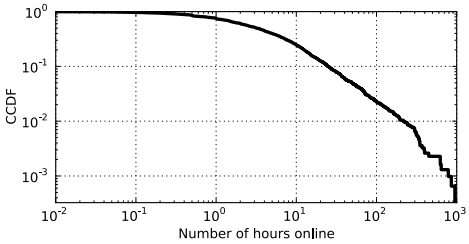


Figure 5: *Session time distribution of Dasu clients (time between their joining and leaving the system).*

the overall stability of the platform, for the same month of January 2013, we saw a total of 1,303 installs, 61 user uninstalls and 21 users who disabled reporting while continuing to run Dasu.

Next, we analyze how the duration of experiments is limited by client session times. Session time is defined as the elapsed time between it joining the network and subsequently leaving it. The distribution of clients' session times partially determines the maximum length of the measurement tasks that can be "safely" assigned to Dasu clients. Figure 5 shows the complementary cumulative distribution function of session times for the studied period. The distribution is clearly heavy-tailed, with a median session time for Dasu clients of 178 minutes or ≈ 3 hours.

Given an average session time, the fraction of tasks that are able to complete depends on the duration of the

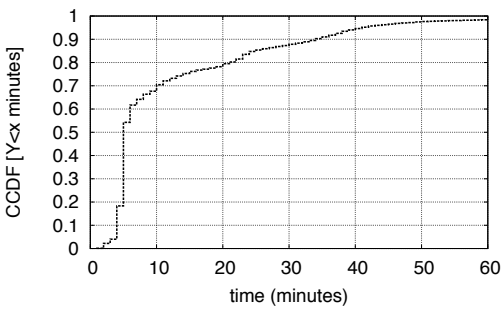


Figure 6: *Task time distribution for completed tasks by Dasu clients. The median task successfully completes in <5 minutes.*

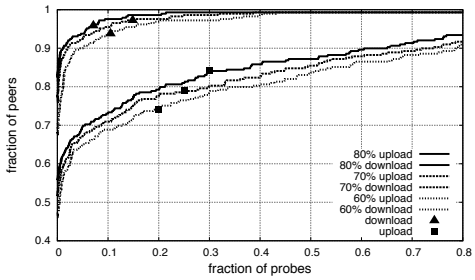


Figure 7: *Distribution of fraction of probes per peer that are delayed due to bandwidth constraints at the client.*

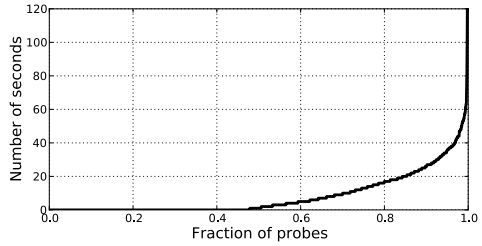


Figure 8: *Distribution of experiment probe submission for clients. Over 55% are launched <1sec. after being scheduled.*

task – a function of the number of actual measurements and the load at the client. Figure 6 shows the distribution of task completion times for all experiments completed by Dasu peers over a 3-week period. All experiments during this period were done in the context of the case study on IXP mapping (Sec. 5), were an experiment task consists of a set of traceroutes issued by clients to discover potential peerings. The figure shows that the median task is able to successfully complete in less than 5 minutes. The plot also shows that nearly all tasks are able to complete successfully in the face of churn, with 70% of tasks finishing in less than 12 minutes.

4.3 Controlling Experimentation Load

To minimize Dasu's impact on host application performance and to ensure that user interactions do not interfere with scheduled measurements, Dasu enforces pre-defined limits on the number of probes executed per unit time and schedules measurements during low utilization periods. We evaluate the impact of one of these restrictions (on bandwidth utilization) on experiment execution by determining the portion of scheduled measurements delayed.

Figure 7 shows a CDF of the fraction of probes delayed by clients due to different bandwidth utilization constraints (60%, 70% and 80%), taken from a random subset of clients over a two-week period. The distribution shows, for instance, that capping at a download utilization of 80%, every scheduled probe can be launched immediately for 85% of the peers, and that for 98% of

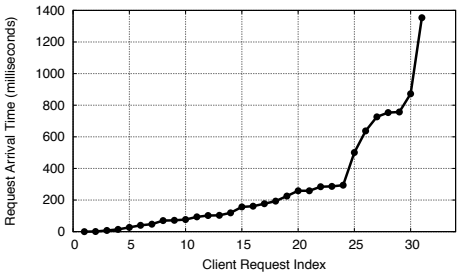


Figure 9: *Request arrival times at the target server. Approximately 80% of requests arrive within 300 ms.*

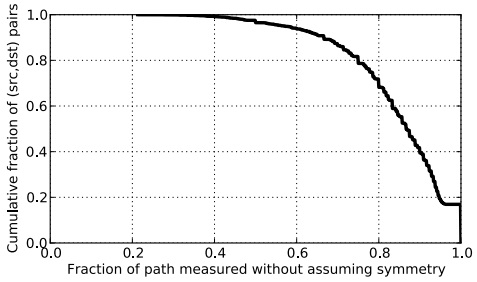


Figure 10: *CCDF of fraction of Dasu-PL path hops that can be directly measured using IP Options probes. 17% of paths reply to probes at each hop, meaning that we can determine the complete reverse path.*

the peers less than 20% of the probes would require any delay. In contrast, a smaller fraction of probes (60%) experience no delay when an 80% utilization limit is imposed on the upload direction. This is expected, since broadband users are often allocated lower upload bandwidth than download.

Fig. 8 shows the queueing time of probes assigned to Dasu clients for a given experiment over a 1-week period. The figure shows that over 55% of the probes are launched in less than a second after being scheduled.

4.4 Client Synchronization

To evaluate the granularity of Dasu's fine-grain synchronization capabilities, we run an experiment where Dasu clients were instructed to simultaneously launch an HTTP request to an instrumented web server. For a span of five minutes, approximately 30 clients were recruited to cooperate in the experiment. Following Ramamurthy et al. [34], as clients joined the experiment they were instructed to measure their latency to the target server as well as to the Coordination Server and to report back their findings.

At the end of the five minutes, clients were scheduled to launch their measurements (having adjusted each request based on their measured latencies) while we logged the arrival times of each incoming HTTP request at the target server. We repeated this experiment 10 times. Figure 9 shows the mean arrival time of each request with a crowd size of 31 clients. About 80% of the requests arrive within 300ms of each other, and 91% of the requests arrive within 1s of each other. This result is on par with the synchronization of 100s of milliseconds reported by Ramamurthy et al. [34]

Variations in the arrival times of the top 20% of requests are due to queuing delays in broadband networks [16] and errors in estimating the latency between clients and the coordinator server.

5 Case Studies

In this section, we present three case studies that illustrate the unique perspective our edge-based platform brings to Internet measurement and serve as examples of experiments made possible using Dasu.

5.1 Extending Earlier Experiments: Routing Asymmetry

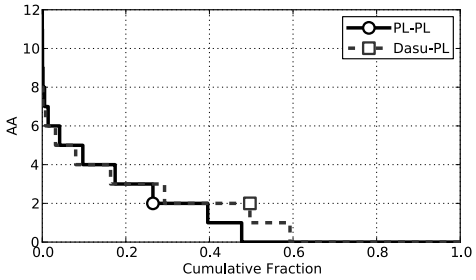
Routing asymmetry can impact the results of measurement tools such as traceroute. For instance, estimates of delay between hosts are subject to errors if the forward and reverse path differ. We extend the work by He et al. [21], comparing routing asymmetry for research and commercial networks, by examining the paths between stub (Dasu) and research (PlanetLab) networks.

Ideally, one would like to control the hosts at both ends of a path to determine the forward and reverse paths between them, and have both endpoints probe the path concurrently to minimize the impact of factors such as network load or time-of-day on routing decisions. The Reverse Traceroute system [24] provides a useful approach to determine the reverse path even when one controls only one of the end points. The approach, however, is not always effective as it cannot probe reverse paths in networks where routers do not reply to IP Options probes.

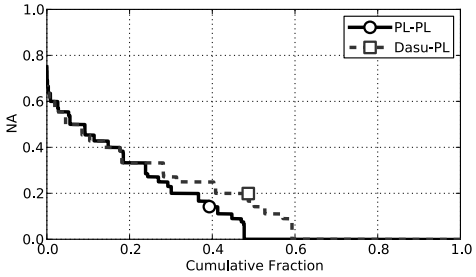
A number of features of Dasu make it possible to conduct an accurate analysis of path asymmetry between nodes located in stub networks vs. research networks including the ability to schedule experiments and to synchronize the launching of measurements across nodes.

We find that for ≈28% of the paths tested between Dasu clients and PlanetLab nodes (out of 8,046) reverse traceroute would be forced to make an incorrect symmetry assumption because a segment of the reverse path transits at least one AS that does not appear on the forward path and that does not respond to IP Options probes. Figure 10 shows that only 17% of the paths between Dasu and PlanetLab nodes respond to IP Options probes at every hop, this in contrast to the over 40% of paths between PlanetLab nodes reported in [24].

To study routing asymmetry between pairs of Dasu-PlanetLab (Dasu-PL) and PlanetLab-PlanetLab (PL-PL)

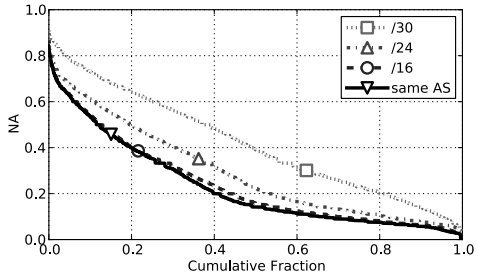


(a) Absolute Asymmetry

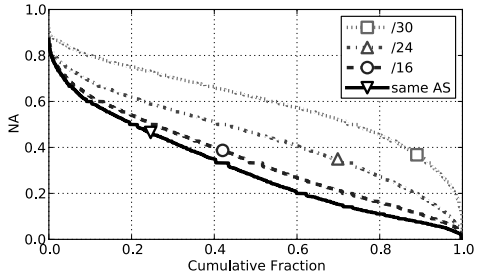


(b) Normalized Asymmetry

Figure 11: *CDFs of AS-level asymmetry in Dasu-PL and PL-PL paths; ≈60% of Dasu-PL paths show some degree of asymmetry, vs. 48% of PL-PL paths.*



(a) PlanetLab-PlanetLab



(b) Dasu-PlanetLab

Figure 12: *CDFs of link-level normalized asymmetry using different heuristics for IP to link mapping. Link-level NA is much lower for PL-PL paths than Dasu-PL paths.*

nodes, we launched probes across 8,046 paths between Dasu clients and PlanetLab nodes, and across 10,067 paths between two PlanetLab nodes. To ensure accurate measures of routing asymmetry we had hosts at both endpoints probe the path concurrently.

We measure routing asymmetry by following the methodology described in [21]. This method maps hops in the forward path to those of the reverse path (either at the link-level or AS-level) and assigns a value of 0 if the hops are identical and a value of 1 if they are different. Through dynamic programming, it then selects the mappings for each path that results in the minimal distance. The minimal composite dissimilarity between a forward and reverse path is referred to as the *Absolute Asymmetry (AA)*, while the length-based *Normalized Asymmetry (NA)* is defined as AA normalized by the length of the round-trip path.

To compare the asymmetry in the AS-level paths between the two sets of paths (i.e., Dasu-PL and PL-PL), figures 11a and 11b show the cumulative distributions of the AS-level AA and NA metrics, respectively. It can be observed that the Dasu-PL paths not only have a higher percentage of asymmetric routes, but also display a higher magnitude of asymmetry than the PL-PL paths. To compare the two datasets at the link-level, we again follow the approach described in He et al. [21] and use their heuristics to determine if two IP addresses correspond to the interfaces of the same link. These heuristics consider two IP addresses to belong to the same point-to-point link if they belong to the same /30, /24, /16, or AS. For each of the four heuristics, Figures 12a and 12b show the cumulative distributions of the resulting NA metric for the PL-PL and Dasu-PL paths, respectively. As noted in [21], the first (/30) and last (AS) heuristics provide the upper and lower bounds, on the observed Internet routing asymmetry at the link level. While figures 11a and 11b show that the two sets of paths exhibit differences in routing asymmetry at the AS-level, figures 12a and 12b show these differences are significantly more pronounced at the link-level but depend greatly on the heuristics used.

5.2 Questioning Existing Experiments: Inferring AS-level Connectivity

The model of the Internet as a hierarchically-structured or "tiered" network of networks is changing [14, 18, 26]. The emergence of new types of networks (e.g., content providers, web hosting companies, CDNs) and their resulting demands on the Internet have induced changes in the patterns of interdomain connections; however, the precise degree and nature of these changes remains poorly understood.

Internet exchange Points (IXPs) are an important part of the rapidly developing Internet ecosystem because they facilitate the changes, enabling direct connections between member ASes. A recent study of a large European IXP has shown that some of the largest IXPs (e.g., DEC-IX and AMS-IX) handle traffic volumes that

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	prefixes
x	*x*	-	-	-	-	-	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*	-	*x*	*x*	*A*
✓	✓	✓	✓	✓	-	-	-	-	✓	✓	✓	✓	✓	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	*B*

Table 4: *Prefix-based peering at Amsterdam Internet Exchange (AMS-IX) between two ASes. Columns show the hour, local time. Legend: '✓' probes crossed IXP; '*x*' probes did not cross IXP; '-' no probes.*

are comparable to those carried by some of the largest global ISPs and support peering fabrics consisting of more than 60% of all possible peerings among their 400-500 member ASes [2]. However, despite their importance, there exists little to no publicly available information about who is peering with whom nor about the nature of these peerings.

These changes in the network's structure demand changes to how we have traditionally conducted experiments. For instance to question the standard assumption of homogeneity that has been made when inferring AS-level connectivity at IXPs [4, 22, 44] – where a single traceroute between two ASes members of an IXP is sufficient to declare that these ASes are, as a whole, connected in the AS graph by a peer-peer link – we require an endemic population of vantage points that allows for finer-grained measurements.

Dasu provides an ideal platform to examine the validity of such assumptions. Its widespread and diverse user base provides vantage points in multiple prefixes within the same AS which allows us to identify prefix-specific features that could not be identified from a single location in the network. Additionally, Dasu's near-continuous availability of vantage points allows us to study temporal effects that are critical for the observed kind of peering. Lastly, conducting this kind of targeted experiments involving specific prefixes in specific ASes at particular IXPs relies critically on the programmability of Dasu.

To evaluate the validity of this homogeneity assumption, we set up an experiment to launch multiple traceroute probes, between the same pair of member ASes of a given IXP, from vantage points located inside different prefixes of the source AS and at different hours of the day. We found that about 15% of the peering links that Dasu discovered violated the assumed homogeneity condition. Depending on the prefixes, the probes either crossed the given IXP or were sent instead via one of the source AS's upstream providers.[7] Table 4 shows a concrete example of such fine-grained peering observed between two ASes at AMS-IX. By probing for peerings between AS1 and AS2 repeatedly from different prefixes in the ASes and separating the probes by the peers' local time, we obtained a view of these well-covered peerings throughout the day. For each data point in the table we corroborated the result across multiple traceroute probes and obtained thus an example of a consistent prefix-based peering – while probes launched from source prefix A towards AS2 are never seen crossing the IXP, probes launched from source prefix B towards AS2 seem to always go through the IXP.

In short, the discovery of such fine-grained or prefix-specific peering arrangements is proof that the traditional view that a single type of AS peering applies uniformly across all prefixes of an AS is no longer tenable. This finding has clear implications for measurement and inference of AS-level connectivity and poses new challenges and requirements for the platforms and techniques used for this type of studies.

5.3 Performing Novel Experiments: Evaluating a Recently-proposed DNS Extension

The *edns-client-subnet* EDNS0 extension (`ECS`) was developed to address the problems raised by the interaction between the DNS-based redirection techniques commonly employed by CDNs and the increasing use of remote DNS services. CDNs typically map clients to replicas on the location of the client's local resolver; since the resolvers of remote DNS services may be far from the users, this can result in reduced CDN performance. `ECS` aims to improve the quality of CDN redirections by enabling the DNS resolver to provide partial client location (i.e. client's IP prefix) directly to the CDN's authoritative DNS server. `ECS` is currently being used by a few public DNS services (e.g., Google DNS) and CDNs (e.g. EdgeCast) and can improve CDN redirections without modifications to end hosts.

To understand the performance benefits of the proposed `ECS` extension and capture potential variations across geographic regions would require access to a large set of vantage points. These vantage points should be located in access networks around the world and allow issuing the necessary interrelated measurement probes. These are some of the unique features that Dasu offers.

Dasu's extensibility allows for the creation and addition of a new probe module to generate and parse ECS-enabled DNS messages. Additionally, Dasu's user base allows us to obtain representative measurement samples from diverse regions and compare trends across geographic areas by looking at the relationships between raw CDN performance, relative proportions of clients affected by the extension, and the degree of performance improvement provided by the extension.

This experiment extends the work by Otto et al. [30], which examined the impact of varying the amount of

[7] The various reasons for why certain ASes engage in such non-traditional peering arrangements is beyond the scope of this paper.

information shared by ECS (i.e. prefix length) and compared its performance to a client-based solution. We first obtain CDN redirections to edge servers both with the ECS extension enabled and disabled. Specifically, we query Google DNS (8.8.8.8) for an EdgeCast hostname. To obtain a redirection with ECS disabled, our DNS probe module sends a query with the ECS option that specifies 0 bytes of the client's IP prefix—this effectively disables the extension's functionality. For the ECS-enabled query, we provide the client's /24 IP prefix. After obtaining CDN edge server redirections with and without ECS's help, we conduct HTTP requests to both sets of CDN edge servers to measure the application-level performance in terms of latency to obtain the first byte of content. For the results from each client, we compare the median performance with and without ECS being enabled.

We analyze results from a subset of 1,185 Dasu clients that conducted this experiment over a 4 month period from September 12th, 2011 to January 16th, 2012.[8] Figure 13 shows the relationship between HTTP latency with ECS disabled and the performance benefits (latency savings) with ECS enabled. We classify users by geographic region; the percentages listed in the legend indicate the fraction of all sampled clients from that region. In all regions, sampled clients are located in a diverse set of networks; even in Oceania—the region with fewest clients—we cover 9 ISPs in Australia and 4 in New Zealand. The figure plots the subset of samples in which EDNS impacted HTTP performance.

While we find clients in all these regions that obtained HTTP performance improvements with ECS enabled, the samples tend to cluster by region. Although clients in North America and Western Europe both typically see HTTP latencies between 20 and 200 ms, the North American clients generally obtain higher percentage savings. This would indicate that the CDN's infrastructure in North America is relatively dense in comparison to that of the public DNS service's deployment. Clients in Oceania typically have relatively high HTTP latencies between 200 and 1000 ms with ECS disabled—but commonly realize savings of 70–90% with ECS enabled. This is likely a result of the specific deployments of the CDN and DNS services; although there are actually CDN edge servers near to clients in this region, it appears that the nearest Google DNS servers are farther away, resulting in reduced HTTP performance when ECS is disabled. Finally, we compare the number of clients with benefits from ECS between Eastern Europe and Oceania; while clients in Oceania actually comprise a slightly smaller fraction of the overall sample, the number of clients that actually observed better performance is much higher than for clients in Eastern Europe.

[8]Each participating client runs the experiment once over that time.

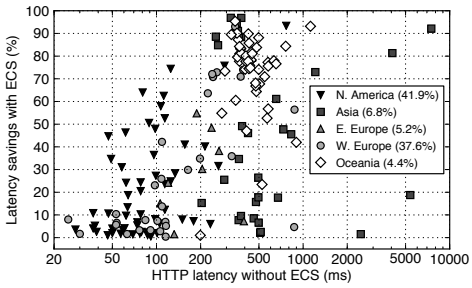


Figure 13: *HTTP latency vs. the performance benefits provided by ECS, by geographic region. Percentages in the legend indicate the geographic composition of the dataset.*

6 Conclusion

We presented Dasu, a measurement experimentation platform for the Internet's edge that supports and builds on broadband characterization as an incentive for adoption. We described Dasu's design and implementation and used our current deployment to demonstrate how participating nodes collectively offer broad network coverage, high availability and fine-grained synchronization to enable Internet measurement experimentation.

Dasu represents but a single point in a large design space. We described our rational for our current design choices, but expect to revisit some of these decisions as we learn from our own and other experimenters' use of the platform.

We presented three case studies that demonstrate Dasu's capabilities and illustrate the unique perspective it brings to Internet measurement. As part of ongoing work, we are exploring the use of node availability prediction for experimentation, approaches to ensure the integrity of experimental results, and allowing fine-grained control of experiments by end users.

The Dasu client is open source and available for download from http://azureus.sourceforge.net/plugin_details.php?plugin=dasu.

Acknowledgements

We would like to thank Lorenzo Alvisi, our shepherd Rebecca Isaacs, and the anonymous reviewers for their invaluable feedback. We are always grateful to Paul Gardner for his assistance with Vuze and the users of our software. This work was supported in part by the National Science Foundation through Awards CNS 1218287, CNS 0917233 and II 0855253 and by a Google Faculty Research Award.

References

[1] Broadband & IPTV progress report. http://www.broadband-forum.org/news/

download/pressreleeases/2012/BBF_IP&TV_Presentation12.pdf, March 2012.

[2] B. Ager, N. Chatzis, A. Feldmann, N. Sarrar, S. Uhlig, and W. Willinger. Anatomy of a large european ixp. In *Proc. of ACM SIGCOMM*, 2012.

[3] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proc. ACM SOSP*, 2001.

[4] B. Augustin, B. Krishnamurthy, and W. Willinger. IXPs: mapped? In *Proc. of IMC*, 2009.

[5] BGP4. IPv4 Looking Glass Sites. http://www.bgp4.net/wiki/doku.php?id=tools:ipv4_looking_glasses.

[6] Z. S. Bischof, J. S. Otto, M. A. Sánchez, J. P. Rula, D. R. Choffnes, and F. E. Bustamante. Crowdsourcing ISP characterization to the network edge. In *Proc. of W-MUST*, 2011.

[7] Caida. Ark. http://www.caida.org/projects/ark/.

[8] M. Casado and T. Garfinkel. Opportunistic measurement: Spurious network events as a light in the darkness. In *Proc. of HotNets*, 2005.

[9] K. Chen, D. R. Choffnes, R. Potharaju, Y. Chen, F. E. Bustamante, D. Pei, and Y. Zhao. Where the sidewalk ends: extending the Internet AS graph using traceroutes from P2P users. In *Proc. ACM CoNEXT*, 2009.

[10] D. R. Choffnes and F. E. Bustamante. Pitfalls for testbed evaluations of Internet systems. *SIGCOMM Comput. Commun. Rev.*, April 2010.

[11] D. R. Choffnes, F. E. Bustamante, and Z. Ge. Crowdsourcing service-level network event monitoring. In *Proc. of ACM SIGCOMM*, 2010.

[12] D. R. Choffnes, M. A. Sánchez, and F. E. Bustamante. Network positioning from the edge: an empirical study of the effectiveness of network positioning in P2P systems. In *Proc. IEEE INFOCOM*, 2010.

[13] A. Dhamdhere and C. Dovrolis. Ten years in the evolution of the Internet ecosystem. In *Proc. of IMC*, 2008.

[14] A. Dhamdhere and C. Dovrolis. The Internet is flat: modeling the transition from a transit hierarchy to a peering mesh. In *Proc. ACM CoNEXT*, 2010.

[15] M. Dischinger, A. Haeberlen, I. Beschastnikh, K. P. Gummadi, and S. Saroiu. SatelliteLab: Adding heterogeneity to planetary-scale network testbeds. In *Proc. of ACM SIGCOMM*, 2008.

[16] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu. Characterizing residential broadband networks. In *Proc. of IMC*, 2007.

[17] FCC. Broadband network management practices – en banc public hearing, February 2008. http://www.fcc.gov/broadband_network_management/hearing-ma022508.html.

[18] P. Gill, M. Arlitt, Z. Li, and A. Mahanti. The flattening Internet topology: natural evolution, unsightly barnacles or contrived collapse? In *Proc. of PAM*, 2008.

[19] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. ACM SOSP*, 1989.

[20] K. P. Gummadi, H. V. Madhyastha, S. D. Gribble, H. M. Levy, and D. Wetherall. Improving the reliability of Internet paths with one-hop source routing. In *Proc. USENIX OSDI*, 2004.

[21] Y. He, M. Faloutsos, S. Krishnamurthy, and B. Huffaker. On routing asymmetry in the Internet. In *Proceedings of IEEE Globecom*, 2005.

[22] Y. He, G. Siganos, M. Faloutsos, and S. Krishnamurthy. Lord of the links: a framework for discovering missing links in the Internet topology. *IEEE/ACM Transactions on Networking*, 2009.

[23] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for Internet-scale systems. In *Proc. USENIX ATC*, 2010.

[24] E. Katz-Bassett, H. V. Madhyastha, V. K. Adhikari, C. Scott, J. Sherry, P. Van Wesep, T. Anderson, and A. Krishnamurthy. Reverse traceroute. In *Proc. of USENIX NSDI*, 2010.

[25] Keynote. Internet health report. http://internetpulse.net/.

[26] C. Labovitz, S. Iekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian. Internet inter-domain traffic. In *Proc. of ACM SIGCOMM*, 2010.

[27] J. Ledlie, P. Gardner, and M. Seltzer. Network coordinates in the wild. In *Proc. of USENIX NSDI*, 2007.

[28] H. M. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: an information plane for distributed systems. In *Proc. USENIX OSDI*, 2006.

[29] MLabs. Network diagnostic tool. http://www.measurementlab.net/run-ndt/.

[30] J. S. Otto, M. A. Sánchez, J. P. Rula, and F. E. Bustamante. Content Delivery and the Natural Evolution of DNS: remote dns trends, performance issues and alternative solutions. In *Proc. of IMC*, 2012.

[31] PlanetLab. PlanetLab. http://www.planet-lab.org/.

[32] H. Pucha, Y. C. Hu, and Z. M. Mao. On the impact of research network based testbeds on wide-area experiments. In *Proc. of IMC*, 2006.

[33] M. Rabinovich, S. Triukose, Z. Wen, and L. Wang. Dipzoom: The Internet measurements marketplace. In *Proc. IEEE INFOCOM*, 2006.

[34] P. Ramamurthy, V. Sekar, A. Akella, B. Krishnamurthy, and A. Shaikh. Remote profiling of resource constraints of web servers using mini-flash crowds. In *Proc. USENIX ATC*, 2008.

[35] RIPE. RIPE atlas. http://atlas.ripe.net/.

[36] SamKnows. Accurate broadband information for consumers, governments and ISPs. http://www.samknows.com/.

[37] Y. Shavitt and E. Shir. DIMES: Let the Internet measure itself. *SIGCOMM Comput. Commun. Rev.*, 35(5), October 2005.

[38] C. R. Simpson, Jr and G. F. Riley. NETIhome: A distributed approach to collecting end-to-end network performance measurements. In *Proc. of PAM*, 2004.

[39] N. Spring, L. Peterson, A. Bavier, and V. Pai. Using PlanetLab for network research: Myths, realities and best practices. *ACM SIGOPS Op. Sys. Rev.*, 2006.

[40] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: a public Internet measurement facility. In *Proc. USENIX USITS*, 2003.

[41] A.-J. Su, D. Choffnes, A. Kuzmanovic, and F. Bustamante. Drafting behind Akamai: Travelocity-based detouring. In *Proc. of ACM SIGCOMM*, Sep. 2006.

[42] S. Sundaresan, W. de Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescapè. Broadband internet performance: a view from the gateway. In *Proc. of ACM SIGCOMM*, 2011.

[43] Vuze. Vuze. http://www.vuze.com/.

[44] K. Xu, Z. Duan, Z.-L. Zhang, and J. Chandrashekar. On properties of Internet exchange points and their impact on AS topology and relationship. In *NETWORKING*. 2004.

[45] R. Zhang, C. Tang, Y. C. Hu, S. Fahmy, and X. Lin. Impact of the inaccuracy of distance prediction algorithms on Internet applications: an analytical and comparative study. In *Proc. IEEE INFOCOM*, 2006.

πBox: A Platform for Privacy-Preserving Apps

Sangmin Lee, Edmund L. Wong, Deepak Goel, Mike Dahlin, and Vitaly Shmatikov

The University of Texas at Austin

Abstract

We present πBox, a new application platform that prevents apps from misusing information about their users. To strike a useful balance between users' privacy and apps' functional needs, πBox shifts much of the responsibility for protecting privacy from the app and its users to the platform itself. To achieve this, πBox deploys (1) a sandbox that spans the user's device and the cloud, (2) specialized storage and communication channels that enable common app functionalities, and (3) an adaptation of recent theoretical algorithms for differential privacy under continual observation. We describe a prototype implementation of πBox and show how it enables a wide range of useful apps with minimal performance overhead and without sacrificing user privacy.

1 Introduction

On mobile platforms such as iOS and Android, Web browsers such as Google Chrome, and even smart televisions such as Google TV or Roku, hundreds of thousands of software apps provide services to users. Their functionality often requires access to potentially sensitive user data (e.g., contact lists, passwords, photos), sensor inputs (e.g., camera, microphone, GPS), and/or information about user behavior.

Most apps use this data responsibly, but there has also been evidence of privacy violations [2, 36, 43, 54, 56]. Corporations often restrict what apps employees can install on their phones to prevent an untrusted app—or a cloud provider that an app communicates with—from leaking proprietary information [11, 28].

There is an inherent trade-off between users' privacy and apps' functionality. An app with no access to user data (e.g., one running in Native Client [39]) cannot leak anything sensitive, but many apps cannot function without such data. For example, a password management app needs access to passwords, an audio transcription app needs access to the recordings of user's speech, etc.

Existing confinement mechanisms deployed on platforms such as iOS and Android rely on users to explicitly grant permissions to apps. In theory, users can decide how much privacy to sacrifice for functionality. In practice, permissions are very coarse-grained (e.g., an app that has permission to access the network can send out whatever it wishes to whomever it wishes), and apps often request more permissions than they need [19, 25] and use granted permissions in unexpected ways (e.g., an app with permission to show the user's location on a map may transmit this location to other parties). Users—who are inundated with permission requests and may not fully understand the implications—often blindly grant all requests [20] or even disable notifications [37], implicitly entrusting all apps with their private data.

Our contributions. This paper describes πBox, a new platform for confining untrusted apps that balances apps' functional needs against their users' privacy, largely preserving both. To achieve this balance, πBox isolates each user's instance of an app from the other instances and users, and only allows communication through a few well-defined channels whose functionality meets the needs of many apps. Because these channels are controlled by πBox, πBox can give rigorous privacy guarantees about the information that flows through them.

The key idea behind πBox is to shift much of the responsibility for protecting user privacy from the apps to the platform. We use three novel technical mechanisms:

1. A sandbox that spans a user's device and a cloud back-end. The latter may be supplied by the device's platform provider (e.g., Apple or Google) or another entity (e.g., the user's employer).
2. Five specialized storage and communications systems that enable a variety of apps to do useful work within πBox while preserving user privacy.
3. An adaptation and implementation of *differential privacy under continual observation* that improves the trade-off between accuracy and privacy of released statistics (e.g., ad impression counts).

Because πBox's sandbox spans the device and the cloud, πBox can help enterprises deploy **bring-your-own-app** (BYOA) policies that allow users to execute apps from untrusted publishers on a trusted platform. This platform may run on the premises under the enterprise's direct control or be part of an external "app store" or hosting infrastructure. Similar to bring-your-own-device (BYOD) policies, where companies install profiles and security software on employee-owned devices used for work, a company might restrict apps to run only within πBox, thus ensuring that these apps—and any information they access—are securely confined.

This paper addresses three research questions raised by this architecture. Can we construct useful apps under these constraints? Can we adapt differentially private aggregation to an environment where app providers need to query periodically updated statistics of user activities? Are the overheads of πBox acceptable?

To answer these questions, we constructed (1) a prototype of πBox and (2) a set of sample apps that represent common app types and demonstrate the utility of our platform: a cloud-backed password vault, an ad-supported news reader, and a transcription service. We also ported two open-source Android apps: the OsmAnd navigation app [41] and ServeStream, an HTTP-streaming media player and media server browser [51]. In Section 2.5, we explain in more detail the classes of apps and app features supported by πBox.

πBox uses differential privacy to prevent aggregate statistics from leaking too much information about users to app publishers. Conventional differentially private queries on static datasets can be very inaccurate when the input data is changing due to user behavior. Instead, we apply algorithms for differential privacy under continual observation [16]—in particular, delayed-output counters. We also list the parameters that enable an app publisher to tune the amount, frequency, and/or accuracy of the reported statistics subject to the platform's bound on the rate of information leakage. The resulting relative error rates on real-world traces are five times lower than with conventional differentially private counters.

The paper proceeds as follows. Section 2 presents an overview of πBox's design. Section 3 shows how πBox deploys differential privacy under continual observation and privacy-preserving top-K lists to implement aggregate channels. Section 4 describes our prototype implementation. Section 5 evaluates it and describes the apps we developed or ported to πBox. Section 6 discusses related work. Section 7 concludes.

2 Design

πBox is a platform for executing apps and associated remote services. There are three types of principals involved in πBox: (1) the platform *provider* who supplies the client (either software, e.g., Google Chrome, or both hardware and software, e.g., Apple iPhone, Google Nexus 7, or Kindle Fire), as well as the cloud resources on which app instances execute, and deploys πBox on both the client and the cloud; (2) *users* who invoke and use untrusted apps on their local devices and their slice of the cloud; and (3) *publishers* who provide apps, content for apps, and/or advertisements.

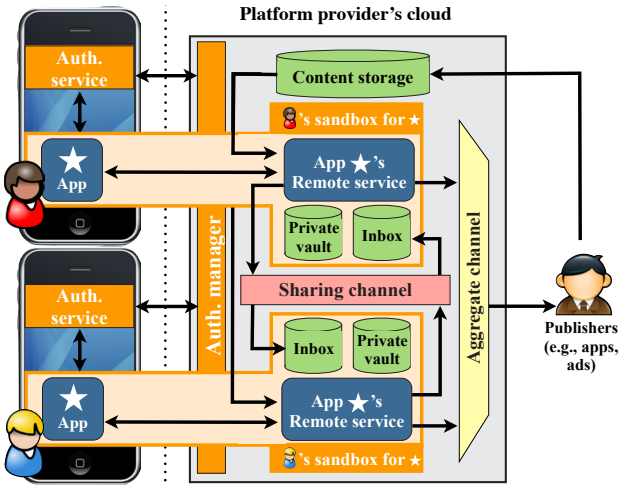


FIGURE 1—Architecture of πBox.

2.1 Threat model

πBox is based on the following design philosophy: *do not trust the apps nor rely on the users to make fine-grained privacy decisions; instead, trust the platform to enforce privacy*. We argue that trusting the platform provider is far more reasonable than expecting users to judge the trustworthiness of many different, often obscure app publishers. After all, users must already trust the platform provider to not leak their private data. Furthermore, third-party platform providers are often trusted brands such as Google, Apple, and Amazon that have strong incentives to take care of their customers' data. Therefore, we assume that both users and app publishers trust the platform, but users do not trust the publishers. Furthermore, we neither assume that the provider trusts the publishers, nor rely on auditing by the provider to eliminate misbehaving apps.[1]

πBox is thus designed for the scenario where *an untrusted app runs in a trusted sandbox*. In this model, the app's publisher may be malicious, the code of the app may attempt to leak users' private data or reveal information about its users to the publisher, some of the app's users may be colluding with the app in an attempt to learn other users' data, etc. That said, the attacker is subject to

[1] Platforms that do audit apps such as Google Play provide additional assurance that is complementary to what πBox provides.

standard computational feasibility constraints (e.g., the attacker cannot subvert cryptographic primitives).

The sandbox provided by πBox is assumed to be trusted. This includes both the components running on the client device and those running in the cloud. Like any software, if πBox is implemented incorrectly, it may be subject to code injection and other attacks that compromise the "ideal sandbox" abstraction. These attacks are outside the scope of this paper, which focuses primarily on the design of the sandbox. Another way in which the "ideal sandbox" abstraction may be violated is via covert (e.g., timing) channels between processes running in the sandbox and those outside the sandbox [33, 47]. If an implementation of πBox is vulnerable to such channels, apps may be able to exfiltrate private data.

There has been much research on sandboxing mechanisms (e.g., [27, 31, 60], among others). This work is orthogonal and complementary to the design of πBox and can be applied to any implementation thereof.

2.2 Extended sandbox

Apps in πBox have two halves: one runs locally on the user's device, the other (optional) runs remotely in the cloud. πBox, executing as the platform both on the device and in the cloud,[2] supplies a per-user, per-app sandbox that spans the device and the cloud. In effect, πBox provides the abstraction that a slice of the cloud is part of the user's device: all of the app's computations and storage are done within this "distributed" device, which is otherwise isolated to protect the user's privacy.

The local half of an app running on the user's device can only connect to the remote half associated with the same app and user. The local half does so by making a request to the *authentication service* running as part of the platform on the device. This service sends the user's credentials and the app's ID to the *authentication manager* running as a part of the platform in the cloud (see Figure 1). Upon successful authentication, the authentication manager starts up the requesting app's remote half for that specific user and opens a secure channel between the local and remote halves.

2.3 Storage and communication

An app running within πBox cannot write data or establish network connections outside of the sandbox. To support app functionality, πBox provides **five restricted storage and communication channels** (see Table 1).

The *private vault* provides per-sandbox (i.e., per-user, per-app) storage that lets an app instance store data specific to a particular user (e.g., user profile, location, query history, etc.) in order to provide personalized services. For example, a password app may use the vault to store the user's passwords, while a news reader app may store keywords of the articles the user has read. Each sandboxed app instance has read/write access to its own private vault; no one else has any access rights.

	Written by	Read by	Purpose
Shared channels for all users of an app			
Content storage	Publisher	App	Store app data and content
Aggregate channel	App	Publisher	Collect usage statistics
Individual channels for each app instance			
Private vault	App	App	App-specific
Inbox	App (via sharing channel), Publisher	App	Receive shared content, notifications from publisher
Sharing channel	App	App (via inbox)	Share content

TABLE 1—Channels in πBox.

The *content storage* provides per-publisher storage for the content that app instances need to function, e.g., maps for a navigation app. Each publisher has read/write access to its own content storage so that the publisher can (1) update the content and (2) grant read-only access to apps that need this content. Apps may draw content from multiple publishers' content storage. For example, an ad-supported news reader may load news articles from a news publisher's storage and ads from an ad broker's storage. Although content storage is shared across all sandboxes that have access to it, read-only access prevents communication between app instances.

The *aggregate channel* provides a per-app channel (shared among all instances of an app) for publishers to collect statistics on users' collective behavior while protecting privacy of individual users. For example, publishers of advertising-supported apps may collect the total number of ad impressions, but not which user viewed which ad. Similarly, publishers of news or video streaming apps may learn which articles or videos are popular, but not who viewed what content. Publishers have read access to their respective aggregate channels, and each app has write access to its channel. In Section 3, we describe how πBox employs differential privacy to protect data released via this channel.

The *inbox* provides per-sandbox storage for the user of a particular app instance to receive information from the app's publisher as well as the content, if any, shared by other users of the same app. Each sandbox has read/write access to its inbox. All writes from the publisher or other users must go through πBox; when publishers want to

[2] Apple (iOS/iCloud) and Google (Android/Cloud Services) already provide app platforms that extend from users' devices to the cloud.

communicate with their apps' users, they submit messages with the user as the recipient, and πBox delivers the message to the appropriate inbox.

Finally, the *sharing channel* provides a per-sandbox method for sharing content with other users of the same app. To ensure that all recipients of the shared content are explicitly approved by the user, we rely on a trusted, platform-controlled dialog box (similar to a "powerbox," which is traditionally used to restrict the paths an app can access [34, 50]). When a user wants to share content from an app, the app writes the data to be shared into its own sharing channel (to which no other sandbox has access) and notifies the platform. πBox controls the rest of the sharing process: it (1) reads in the data, (2) presents the data to the user in a dialog box that explicitly notifies the user about the imminent sharing of the presented data, (3) prompts the user to confirm the recipients, and, upon confirmation, (4) writes the shared content to the inboxes of the designated recipients' sandboxes. This design ensures that users are aware when and with whom sharing occurs, but it cannot prevent the app from surreptitiously leaking private information in the shared data (e.g., through steganography).

2.4 Advertising and third-party services

Advertising. To broadly support free apps, many of which are financed by ads, πBox must support in-app advertising. Traditionally, advertisers tell ad networks which ads to display, how much they are willing to pay per impression, and the interests they are targeting. Ad networks organize ads into lists ranked by factors such as the bid, number of impressions already made, etc. When an app wants to display an ad, the ad network provides an ad based on the user's perceived interests.

To prevent apps from leaking users' private data to advertisers, πBox changes this process: (1) the ad network must store its ads in content storage on the πBox cloud platform, (2) the number of impressions must be released via the aggregate channel (see Section 3.1), and (3) the logic for selecting and fetching an ad from content storage (based on the user's profile, activities, etc.) and the logic for outputting to the aggregate channel must be implemented inside the app (e.g., as part of a SDK or library) and executed inside the sandbox. For efficiency, πBox allows publishers to share content storage across multiple apps. Since apps have read-only access, this does not affect privacy guarantees.

πBox protects users' identities and thus prevents ad networks from singling out individuals who may be engaged in ad impression/click fraud. That said, other defenses [22]—per-user thresholds on the number of impressions/clicks, bait ads, and using historical statistics to detect apps that pad the number of impressions/clicks—continue to be effective even with πBox.

Ads that click-through to external sites can leak a user's identity (or at least the IP address) and other private information.[3] In πBox, arbitrary network traffic out of the sandbox is not allowed, and click-through ads must redirect the user to trusted platform resources, e.g., an ad page in the ad network's content storage.

Although not yet implemented, conventional click-through ads can be supported in πBox with some modifications. First, all click-through URLs must be prespecified and static for all app instances (they cannot be dynamically generated or otherwise based on the information observed by a given instance). This still allows a potential leak because the app's choice of predefined ads to show to the user may depend on the user's private information, but requiring static URLs limits the rate of leakage. Second, the platform must verify that the click indeed originated from the user. To support this, πBox can use a trusted powerbox dialog to prompt the user for explicit consent, similar to the sharing channel, before permitting the click to go through. We believe, however, that this point in the design space for ad support sacrifices privacy, complicates the guarantees provided by πBox, and forces users to make privacy decisions for which they may not fully understand the implications.

πBox does not currently support ad networks that choose which ads to serve via a real-time auction. Such auctions require either that users' profiles be sent to the advertisers (so they know what they are bidding on), or that all bidding logic be part of the sandbox. Alternatively, there exist proposals for privacy-preserving ad auctions [46]. Advertising based on real-time bidding accounts for less than 30% of all advertising sales [45], and the introduction of "Do Not Track" in Web browsers may adversely impact auction-based advertising [17].

Third-party services. Because πBox does not allow apps to communicate outside of the platform, apps cannot use external third-party services such as content delivery networks (CDNs). As with ads, apps running on πBox can only access content and use services that are hosted by the platform provider and published in the read-only content storage. Fortunately, many platform providers already provide services for apps, e.g., maps from Apple, Google, and Bing, or CDN services such as Amazon CloudFront and Google PageSpeed.

2.5 Apps supported by πBox

Figure 2 lists many app features and indicates whether and how πBox protects user privacy for each of them. In general, apps that do not involve sharing between

[3] For example, a set of ads may only be shown to (and thus clicked by) users matching certain criteria or even maliciously micro-targeted to specific individuals [30].

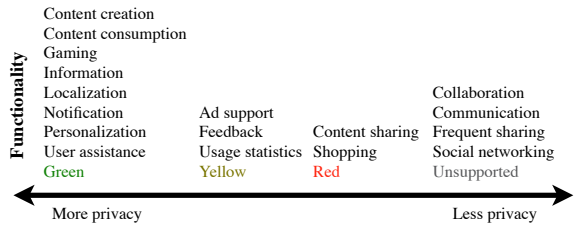


FIGURE 2—πBox support for different app features.

users are well-suited for πBox. This includes, for example, multimedia, reference, weather, and utility apps, many of which handle sensitive data (e.g., navigation, personal finance, password management, malware detection, speech recognition, etc.). πBox supports the reporting of usage statistics, user feedback, and ad impressions.

Some apps only share content occasionally, e.g., games that let users share their scores, or camera apps that let users share some of their pictures. For these apps, πBox protects user privacy with respect to the app's core functionality. Furthermore, the πBox sharing channel ensures that any content sharing is explicitly authorized by the user (malicious apps may still exfiltrate sensitive data by hiding it in shared content—see Section 2.6).

Finally, there are apps—e.g., Facebook, Twitter, or multiplayer online games—whose sole purpose is to allow users to connect, communicate, collaborate, and share content with other users. Users of such apps already expect to lose some of their privacy, and πBox can guarantee relatively little for them.

Each πBox-supported app is assigned a **privacy rating** determined by the channels it uses. Apps that only use the private vault, content storage, or inbox are **green**: they never export any data from the sandbox and cannot leak anything. Apps that use the aggregate channel are **yellow**: they may release differentially private statistics but there is a provable bound on the amount of information leaked. Finally, apps that use the sharing channel are **red**: they rely on explicit user consent to export information and are at a higher risk of leaking private data. In Section 5.4, we describe how many top apps from the Google Play store fall into these categories.

2.6 Limitations and scope

πBox reduces privacy risks to the users of many apps and makes it more difficult to harvest large amounts of private user information, but it is not a privacy panacea.

First, the differentially private aggregate channel leaks a little information with every output. This is inevitable, and we quantify this leakage in Section 3. Note that no covert communication beyond this leakage is possible over the aggregate channel because differential privacy holds regardless of the recipient's auxiliary (including covert) information. In the case of πBox's aggregate channel, the *timing* of the release is differentially private, too, precluding a malicious app from encoding covert information in the timing of its aggregate outputs.

Second, while πBox's sharing channel guarantees that only the specified recipient can read the shared content, a malicious app may hide private information in this content via steganography. Several factors mitigate this risk. First, πBox shows the content to be shared to the user and uses the powerbox mechanism to directly confirm the user's consent to share. Second, πBox restricts the type of content to be shared: only plain text and images are allowed in our prototype.

This is a trade-off between usability and privacy. The design philosophy behind πBox is to avoid involving users in privacy-critical decisions (in contrast to the Android permission system). At the same time, sharing is important for many applications, and πBox lets users explicitly accept a privacy risk when sharing content.

Most importantly, πBox guarantees that shared content can only be viewed by the recipients who have been explicitly approved by the user. While a malicious app instance may be able to embarrass the user by sending private information to an approved recipient, the app publisher still does not have access to this data unless the recipient (or the user who is sharing) cooperates.

In general, we believe that πBox will be appealing to entities looking to (1) enhance or safeguard their existing app platforms by improving user privacy, (2) rent privacy-preserving cloud resources to app publishers, and/or (3) provide a curated version of their standard app store that offers privacy-enhanced apps to enterprise customers. πBox is an especially good fit for enterprise environments, where apps typically contain content from a single external publisher, do not require (in fact, frequently forbid) sharing of content outside the enterprise, do not rely on ads, and do not involve functionalities with multiple external parties such as brokered ad auctions.

3 Protecting privacy

The functionality of many apps depends, both technically and financially, on some information about their users. Aggregate statistics are often sufficient—for example, some ad-supported apps only need to track the number of ad impressions, not whether a particular user viewed a given ad—but even they may reveal information about individuals [8, 14].

πBox uses *differential privacy* [14] to enable app publishers to collect relatively accurate statistics on users' behavior while limiting information leaks about any individual user. Informally, differential privacy is a framework for designing computations where the influence of any single input on the output is bounded, regardless

of the adversary's knowledge and/or external (auxiliary) sources of information the adversary may have access to.

"Conventional" differential privacy techniques such as the Laplacian mechanism (described in the following section) are primarily intended to protect individual inputs in computations on static datasets. By contrast, apps keep generating new data: for example, an app may continuously update the number of times a news article has been read or an ad has been shown. Moreover, app publishers may be interested in rankings, such as the most popular news articles or the most frequently misrecognized words in a transcription app. As we will show, conventional mechanisms, while privacy-preserving, result in an unacceptable loss of accuracy in these settings.

To balance privacy and accuracy, πBox deploys recently developed algorithms for differentially private counters under continual observation [16] and differentially private ranked lists [7]. To the best of our knowledge, πBox is the first system that uses differential privacy under continual observation in a working system.

3.1 Counters and top-K lists

In πBox, the key building block for the aggregate channel is a set of platform-controlled *counters*. As an app executes, it may increment one or more counters. Eventually, the (randomly perturbed) values of these counters are released to the app publisher. The list of counters must be defined by the publisher in advance. Therefore, a malicious app instance cannot encode user-specific information in its choice of counter *names*. The released counter *values* are differentially private and thus probabilistically hide the influence of any given user's data.

πBox enforces *user-level* differential privacy on these counters, i.e., the privacy of all data, actions, and any other inputs associated with a particular user, as opposed to the privacy of a single input. Formally, for some privacy parameter ϵ (described further in Section 3.2), a computation F satisfies user-level ϵ-differential privacy if, (1) for all input datasets D and D' that differ only in a single individual user whose inputs are present in D but not in D', and (2) all outputs $S \subseteq Range(F)$,

$$\Pr[F(D) \in S] \leq e^{\epsilon} \cdot \Pr[F(D') \in S] \qquad (1)$$

A standard mechanism for making any computation F differentially private is the *Laplacian mechanism*, which adds random noise from a Laplace distribution to the output of F before it is released, i.e., $F(x) + Lap\left(\frac{\Delta F}{\epsilon}\right)$. Here $Lap(y)$ is a Laplace-distributed random variable with mean 0 and scale y, and ΔF is the maximum possible change in the value of F (F's sensitivity) when a single user's inputs are removed from the dataset.

Intuitively, the more sensitive a computation is to its inputs, the more random noise is needed to ensure a given level of privacy. Consequently, ΔF in πBox—and, therefore, the amount of noise that πBox adds to the released counter values—depends on the number of counters a user can update (which we denote as n) and the maximum amount by which a user can affect any single counter (s). There is an important trade-off in the Laplacian mechanism between privacy (ϵ) and accuracy: *higher accuracy requires giving up more privacy*. We will revisit this trade-off in detail in Section 3.2.

Parameter chosen by platform provider
Per-period privacy budget (R)
Parameters chosen by app publisher
List of counters (L)
Frequency of output release (f)
Privacy parameter (ϵ)
Max. # counters app instance can update per period (n)
Max. contribution to each counter per period (s)
Buffer size (b)
# of ranked counters (K)

TABLE 2—Parameters for aggregate counters. b and K only apply to delayed-output and top-K counters, respectively.

Supporting periodic updates. Many apps dynamically update counters during execution and then need to periodically release them. The Laplacian mechanism can be applied to every release, but if the timing of releases is independent of the counter's true value, the random noise added by the mechanism (which, too, is independent of the counter's value) can be much larger than the true value, resulting in high relative error. This arises, for instance, when counting the number of impressions for rarely displayed ads targeting a niche group of users.

πBox uses *delayed-output counters* [16] instead. Figure 3 describes how such a counter is implemented. Intuitively, this mechanism randomly delays releases of the counter value; if the value is small relative to the noise that must be added, the release is likely to be postponed.

Furthermore, rather than allowing counters to be continuously queried, πBox enforces a minimum interval between releases (line 5). Thus, even the counters that have internally accumulated a large number of updates may not be immediately released. Delaying the release may affect the freshness of the released values, but the relative error will be smaller.

Supporting ranked top-K lists. To release top-K lists, πBox adapts techniques by Bhaskar et al. [7]. The app publisher specifies K beforehand, and the amount of noise that is added is proportional to K, which is typically smaller than the amount of noise (proportional to n) that would have been added if we had used the Laplacian mechanism on every counter to determine the top K. To generate a ranking of the counters without their associated values, the algorithm adds $Lap(4Ks/\epsilon)$ random noise to the values of all counters and picks the

```
1: V_i : true count in period i
2: λ ⇐ s·n / ε
3: A ⇐ 0
4: D ⇐ b + Lap(λ)
5: for each period i of duration 1/f do
6:    A ⇐ A + V_i
7:    if A − D > Lap(λ) then
8:       Release A + Lap(λ)
9:       A ⇐ 0
10:      D ⇐ b + Lap(λ)
11:   end if
12: end for
```

FIGURE 3—Delayed-output counter.

top K counters based on these noisy values. If an app publisher needs to know the actual values of the associated counters as well, the algorithm adds an additional $Lap(2Ks/\epsilon)$ noise to the true values of the selected K counters before releasing their values.

It may appear that the ability to release top-K lists allows apps to leak sensitive information. For example, the publisher of a password management app could learn the K most common user passwords (in any case, these are already well-known). Note, however, that the publisher cannot learn the password of any given user. Similarly, conventional differential privacy allows the publisher to ask how many users have a particular password, but the answer does not reveal any specific user's password.

Finally, πBox's aggregate channel can be extended to support other differentially private functions such as mean and threshold [48].

3.2 Choosing privacy parameters

Absolute privacy cannot be achieved: as long as the released values have any utility, the original data can be reconstructed after observing at most a linear (in the size of the dataset) number of values [13]. To model the cumulative loss of privacy after multiple computations on the same private data, differential privacy uses the notion of a *privacy budget* [15, 35]. Every ϵ-private computation charges ϵ cost to this budget. The higher the value of ϵ, the less noise is added, thus the released value is more accurate, but the privacy cost is correspondingly higher, too. The budget is pre-defined by the data owner. Once it is exhausted, no further release is allowed.

In our setting, it is undesirable for an app to lose functionality after a while. Instead, πBox enforces a *per-period* privacy budget that bounds *privacy loss per period* by parameter R, which is chosen by the platform provider. For a given R, the app publisher may specify the types of the counters the app will release (delayed-output and/or top-K with or without associated values), as well as the relevant parameters in Table 2, so long as

$$c \cdot f \leq R \tag{2}$$

where $c = \epsilon/2$ for top-K counters without associated values and ϵ for the other two types of counters.

To understand how c and ϵ relate to the amount of information leaked, let P be an adversary's prior probability of the user's private data having a particular value and P' be the posterior probability after observing the released counters. Condition (1) ensures that $P' \leq e^c \cdot P$, i.e., any released value changes the adversary's prior probabilities (no matter what they are!) by no more than a constant multiplicative factor. If uncertainty is measured as min-entropy of the adversary's probability distribution over the private data,[4] every release yields $(c \log_2 e)$ bits of information to the adversary [1, 6]. Given this representation of uncertainty, πBox's counters release at most $(f \cdot c \log_2 e) = (R \log_2 e)$ bits per period. For example, an app that uses delayed-output counters with $\epsilon = 1$ and the release frequency f of once per day leaks at most 1.44 bits of information daily.

While it is straightforward to calculate how much noise should be added for a given choice of counter type and ϵ, the utility of a particular counter arguably depends not just on the amount of noise added, but also the actual true counter value, i.e., the *relative* amount of noise matters. The larger the true value, the larger the absolute noise that can be tolerated for a given relative error, thus allowing for smaller values of ϵ.

As long as condition (2) is followed, app publishers are free to choose the types of the counters used by their apps and the values of the parameters listed in Table 2. For example, a publisher may want more frequent output (f), at the expense of lower ϵ, higher λ and thus lower accuracy. To maintain the same accuracy, the publisher may keep the same λ at the cost of decreasing the maximum number of counters a single app instance can update (n) and/or the maximum amount it can contribute (s).

4 Implementation

We implemented a prototype of πBox using Android 2.3 (Gingerbread) for the device client; Jetty [29], a Java servlet container, for the remote services; and HBase [23] for the cloud communication and storage channels. The trusted computing base (TCB) consists of the above software, cloud operating system (Linux in our case), and the πBox implementation, which itself is approximately 7,500 lines of code for the cloud half and 2,700 for the device half. The design of πBox is largely agnostic to the specific sandboxing technology and could have used

[4] The min-entropy of a probability distribution that assigns probability p_i to some event i is $-(\max_i \log_2(p_i))$.

virtual machines, Native Client [39], or more advanced sandboxes, which would change the size of the TCB.

4.1 Isolation and authentication

Client isolation. To implement the sandbox on the device, we augmented Android's built-in sandboxing mechanism. By default, Android assigns each app a unique user identifier (UID). πBox allows non-privacy-preserving apps to coexist with privacy-preserving apps on the same device, but assigns UIDs from different ranges to apps of different types. This makes isolation enforcement simpler in the kernel code.

Android uses standard Linux permissions to isolate apps from each other, but this is not enough to prevent an app from abusing the permissions it has. To prevent πBox-confined apps from leaking private data, we modify Android to block them from creating world-readable files or directories, and from writing to files or directories owned by another app's UID.[5] πBox does not allow confined apps to communicate with other non-system apps via IPC, including Binder IPC (the basic primitive for various higher-level Android IPC mechanisms). Finally, we use iptables to confine the apps' network traffic. These changes are applied at the kernel level only to πBox-confined apps (recognized by their UIDs).

Cloud isolation. We implement the server-side functionality for πBox apps as Java servlets using Jetty. Many existing Web apps, e.g., those on Google App Engine [4], can thus be easily adapted to πBox.

In Jetty, each app is isolated in a separate Web app context (a container that shares the same Java class loader). In πBox, each user of an app is also isolated in a separate context, achieving classloader-level isolation. To restrict the servlet's communication via system resources, we rely on Java's security monitor. Our sandbox also includes many other restrictions used by Google App Engine, e.g., disallowing reflection and controlling access to JVM-wide resources such as system properties.

Authentication. When an app on the user's device wants to communicate with its cloud-based half, it sends an "intent" (a high-level IPC mechanism in Android) to πBox's local trusted authentication service, implemented as a system app. After identifying the requesting app, the authentication service requests the user's credentials via user input or from a cache and sends them, along with the app's ID, through a TLS tunnel to πBox's authentication manager in the cloud. Upon successful authentication, the authentication manager sets up a new servlet instance at a specific URL, establishes an IPsec endpoint on the machine where the servlet is instantiated, and sends this URL, a one-time password that is required to access the servlet instance, and the IPsec key to the authentication service on the user's device.

The authentication service establishes the other end of the IPsec tunnel on the device, updates iptables to allow the app to communicate with the servlet, and passes, via intent, the URL and password to the app. IPsec ensures that all communication to and from the servlet is encrypted, and iptables ensure that the app on the user's device can only communicate with the user's servlet instance via this IPsec tunnel. Finally, the app running locally on the user's device authenticates using the provided password via HTTP basic authentication over the IPsec tunnel (which encrypts the credentials); this step ensures that only this specific app can communicate with the servlet. Once this process is complete, the app can send HTTP requests to the provided URL and receive HTTP responses from its cloud component.

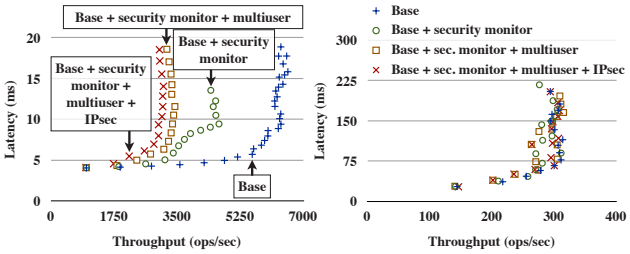
4.2 Storage and communication channels

πBox's storage systems use local device storage and HBase, a popular NoSQL storage system. Local device storage is part of πBox's private vault. Any data that is written to local storage is secured as described in Section 4.1 and cannot be exported from the sandbox. Access to cloud storage is provided via a HBase-like API.

When an app publisher submits an app to the platform, the publisher provides a WAR (Web application ARchive) file that contains the app's servlet code and XML files that describe the schemas of the HBase tables that the app needs for each type of cloud storage. To implement various channels, πBox provides wrappers of the HBase client that expose the appropriate interfaces to servlet instances. For example, the interface to content storage exposes read-only operations on the storage's shared tables. The interface to the cloud-backed private vault provides both read and write access to the per-sandbox table. The wrapper for the aggregate channel exposes an update-only interface for the counters, which are stored in the HBase tables by πBox. Stored counter values are periodically released by (1) sanitizing them via the differential privacy module using the parameters provided by the app publisher (Section 3.2) and (2) writing them to a table that can be read by the publisher.

The per-sandbox inbox allows a user's servlet to receive messages from the app publisher or from another user's servlet for the same app. This inbox is implemented using an HBase table in which each row corresponds to a single message. The row includes the sender's platform username (the name used to authenticate with the authentication service or a special username reserved for the app's publisher), a timestamp, and the message body. Messages from the publisher are de-

[5] This implies that an app can only write to directories that it alone has read access to and that other apps cannot see the files it has written.



(a) Static (b) SHA-256

FIGURE 4—Latency vs. throughput for πBox mechanisms.

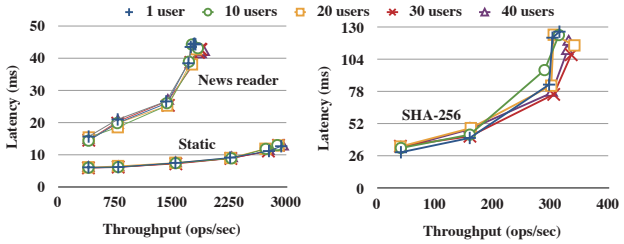
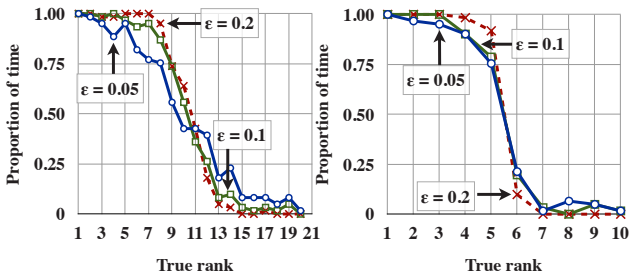


FIGURE 5—Overhead of user isolation for various workloads.



(a) Top-10 (b) Top-5

FIGURE 6—Fraction of time the true top-K documents appear in the noisy top-K list.

livered to the recipient's inbox by a designated servlet, which can be invoked only by the authorized publisher.

Lastly, when an app wants to share content through the sharing channel, it sends an intent, along with the content to be shared, to πBox's sharing service, which is implemented as part of the authentication service. The sharing service prompts the user for the recipients' usernames and sends the message, along with the usernames of the sender and the recipients, to a designated servlet that only the platform can access. This servlet then adds the message to the inbox of each recipient.

5 Evaluation

5.1 Performance overhead

We evaluate πBox using a server with two four-core Xeon E5430 CPUs and 16 GB RAM and 4 clients with a single-core 3 GHz Pentium 4 Xeon CPU with hyper-threading and 1 GB of RAM, all running Fedora 8.

We first use micro-benchmarks to measure the throughput and response time of the various mechanisms employed by πBox on two types of workloads: a simple static workload where the server responds with about 10 bytes of static HTTP body data, and a computationally intensive workload where the server randomly generates 1 MB of data and calculates its SHA-256 hash. We generate the workloads by having a varying number of clients continuously submit requests over a 30-second interval.

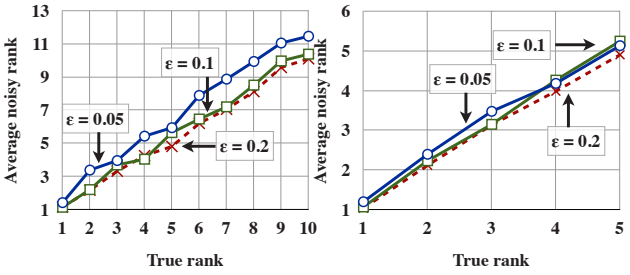
Figure 4 shows the results with different components turned on. In the base configuration, we run the server with the Java security monitor disabled, no isolation (i.e., a single servlet instance serves all client requests), and without an IPsec tunnel between the server and the clients. We then enable the security monitor, run multiple servlet instances to serve different clients, and/or enable IPsec. For the simple static workload, πBox reduces the throughput of the system by roughly 50%, incurring an overhead of 0.17 ms per operation. For the heavier SHA-256 workload, however, the computation required to generate the hash effectively hides the overhead of πBox.

To measure the overhead of isolating app instances, we fix the load offered to the server (i.e., the number of requests generated by the clients) and vary the number of Web app containers (i.e., per-client servlet contexts) on the server. Figure 5 shows the throughput and response time of πBox for three types of workloads, with requests uniformly distributed across the containers. The static and SHA-256 workloads are the same as in the previous experiment. In the news reader workload, clients request a list of new articles (about 300) and a specific article (5 to 10 KB) from the servlet half of our news reader app (Section 5.3). This causes many I/O-intensive operations on the small HBase instance that stores the articles. As Figure 5 shows, the overhead of user isolation is insignificant for all three workload types.

5.2 Privacy vs. accuracy

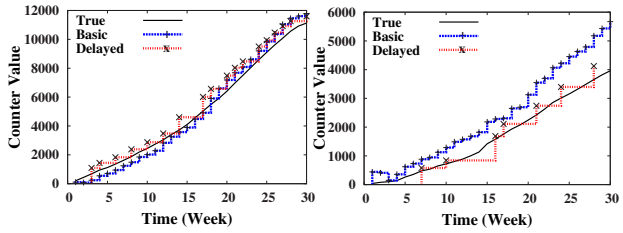
To show that the differential privacy mechanisms employed by πBox provide reasonable accuracy in real-world scenarios, we first apply the top-K mechanism to the 60-day Web server trace of the 1998 World Cup website [59]. For each day, we calculate the top 5 and top 10 most frequently accessed documents and use the πBox's aggregate channel to output "noisy" top-5 and top-10 lists. The total number of daily accesses for a top-10 document ranged from 6,000 to 14,000.

Figure 6 shows, as a function of the privacy parameter ϵ, how often the true top-5 and top-10 documents on a particular day appeared in the noisy, privacy-preserving top-5 and top-10 lists output by the aggregate channel. As ϵ increases, the accuracy of the noisy rank lists improves. For example, the 8th-ranked item appears in the



(a) Top-10 (b) Top-5

FIGURE 7—Average noisy rank for a given true rank.



(a) Frequently accessed (b) Infrequently accessed

FIGURE 8—Accuracy of delayed-output counter on two different documents. We use $\epsilon = 1$, $|L| = 100$, and $b = 500$.

noisy top-10 list 75% of the time when $\epsilon = 0.05$, but 95% of the time when $\epsilon = 0.2$. This percentage is even higher for items with higher true ranks. Figure 7 shows the average noisy rank given to true top-5 and top-10 documents. The accuracy of the noisy rank improves with higher ϵ; with $\epsilon = 0.2$, all ranks are correct.

To illustrate the advantages of the delayed-output mechanism for releasing infrequently updated counters, we use a trace from the University of Saskatchewan Web server [49] which contains a variety of access patterns. For this experiment, we set $\epsilon = 1$, the total number of delayed-output counters ($|L|$) to 100, the buffer size (b) to 500, and the release frequency to 1 week. We compare the delayed-output counter to a basic counter that simply outputs its differentially private value every week. Figure 8 shows the values of the delayed-output counter and the basic counter over a 30-week span for two documents with different access patterns. For the frequently accessed document, the delayed-output counter is off by 12.9% on average vs. 19.6% for the basic counter. For the less frequently accessed document, the delayed-output counter is much more accurate, with a relative error of 15.6% vs. 83.1% for the basic counter.

5.3 Apps

To illustrate how to build useful privacy-preserving apps in πBox, we developed three sample apps and ported two existing open-source apps.

Password manager. A password manager is an example of an app that needs to keep (but not share) sensitive

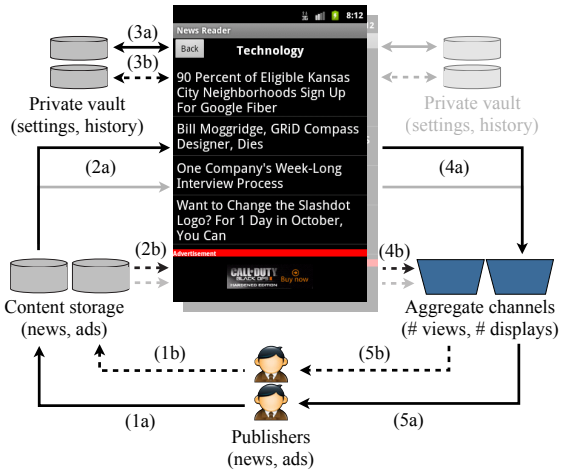


FIGURE 9—Interactions and data flow between the news reader app and πBox. The dark (solid, dotted) lines represent the flow from the (content, ad) publisher. The lighter lines represent the same flows for another user of the same app.

data, e.g., store a user's credentials in the cloud so that the user can access them from different devices and to avoid keeping them on the devices themselves. Although many such apps use encryption, the user must trust that the app's publisher is neither malicious nor incompetent.

Our πBox-based password manager app simply stores the user's passwords in its cloud-backed private vault, enabling their retrieval from multiple devices. Despite its simple design, the app guarantees that (1) only a specific user can access the stored password via the app, and (2) the app cannot leak the stored passwords to anyone else (i.e., this app is "green"; see Section 2.5). This benefits both the user, who does not have to worry about the trustworthiness of the app, and the app publisher, who can rely on πBox to secure the publisher's app's storage.

News reader. Our news reader app is an example of an ad-supported media browsing and consumption app that uses πBox's storage systems and involves multiple publishers. Figure 9 shows the flow of data between the publishers, the app, and the platform.

The main functionality in any news reader app is displaying content (news articles) to the user. In our implementation, the publisher supplies the articles by adding to, updating, and removing from the app's content storage located on πBox's cloud platform (Figure 9, 1a). The app has read-only access to this storage (Figure 9, 2a).

The news reader may provide personalized content to the user, for example, recommend certain articles based on the user's reading history. It can track the user's reading history by writing to its private vault (Figure 9, 3a). Because the vault is per-user and per-app, this data cannot leak to other app instances or the app publisher.

Many apps of this type are ad-supported. The ads may be published by either the app publisher or a separate en-

tity, e.g., an advertising network partnering with the app publisher. Like the news articles, the ads are published and updated by their publisher and viewed by the user via content storage (Figure 9, 1b and 2b). Any personalization and micro-targeting is done by writing the relevant data to the private vault (Figure 9, 3b).

Both news and ad publishers may want to know how often their articles and ads have been viewed. Our app keeps one counter per article and ad. We use a top-10 list to track the most popular articles (Figure 9, 5a) and delayed-output counters for ad impressions (Figure 9, 5b), since the latter do not need to be released frequently.

The news reader app is a "yellow" app: although it exports statistics, πBox provides differential privacy guarantees to its users. It is straightforward to extend the news reader to let users share interesting articles with other users, which would make the app "red."

Transcription. Our transcription app uses cloud-based voice recognition. It records the user's speech on the device and transmits it to a servlet, which writes the recording to per-sandbox temporary scratch space and executes Sphinx-4 [53], an open-source speech recognition toolkit, to transcribe the text. The transcription is then sent back to and displayed by the app on the device. Our current prototype keeps the dictionary in the app's binary but we could also use content storage for this purpose, allowing the publisher to update the dictionary.

This app uses the aggregate channel to release the confidence scores of speech recognition for each l-gram ($l = 1$ in our prototype). First, the app publisher defines counters for all words in the Sphinx-4 dictionary (per Table 2, L is the list of these counters, $n = |L|$). Sphinx-4 provides confidence scores that range from 0 (low confidence) to 1. Because the publisher is likely interested in the most misrecognized words, our app inverts the score (thus making higher scores reflect lower confidence, up to a maximum of $s = 1$) before adding it to the previous value of the counter. The top-K list thus contains the K (10 in our prototype) most misrecognized words.

The transcription app is a "yellow" app. πBox guarantees that, even if the recordings of the user's speech contain highly sensitive data, the app can leak this data only through the differentially private aggregate channel as (noisy) top-K word lists, which do not identify the actual words spoken by specific users.

Porting existing apps. We ported OsmAnd [41], an Android navigation app based on OpenStreetMap [40], and ServeStream [51], an HTTP-streaming media player and media server browser, to πBox.

The major changes to the apps involved (1) adding code to initiate authentication via πBox's authentication service, (2) modifying all HTTP requests app to include the authentication credentials provided by the authentication service (Section 4.1), and (3) moving map and media content into πBox's content storage and serving them via servlets. The use of HTTP as the communication protocol simplified porting these apps to πBox, but this simplification is likely to apply to many other apps. Overall, for OsmAnd, we modified or added 174 out of 119,147 lines of code; for ServeStream, 133 out of 13,193 lines.

Both ported apps use only the private vault and content storage, making them "green."

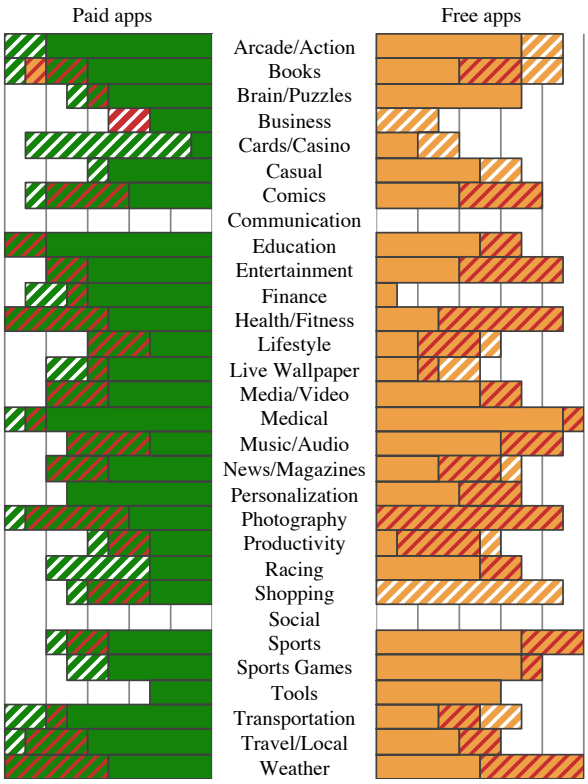


FIGURE 10—Number of top-10 apps in Google Play categories (as of Feb. 2013) that can be supported by πBox. Unsupported apps are uncolored/white. Stripes represent apps that, due to non-core sharing or unsupported functionality, are one color but whose core functionality is another color, e.g., a PDF viewer that allows sharing is red, but its core is green.

5.4 Coverage of existing apps

To further evaluate how well πBox can support existing app functionalities, we surveyed the top 10 free apps and top 10 paid apps from all categories excluding wallpaper, widget, and library in the Google Play app store, for a total of 30 categories and 600 apps. This survey was based solely on the developer's description of the app in Google Play, thus the reported numbers are only estimates.

Figure 10 shows how many apps can be supported by πBox and the degree of support. Among the paid apps, 46% are green, 18% red, and 36% unsupported; considering only core functionality, 74% are green. Among the

free apps, 37% are yellow, 21% red, and 42% unsupported; considering only core functionality, 67% are yellow. Unsurprisingly, many of the unsupported apps are those that are categorized as "communication" or "social" and thus require frequent sharing of data. Free apps are largely ad-supported and thus at least yellow.

6 Related work

xBook [52] and the system of Viswanath et al. [57] employ an extended sandbox mechanism similar to πBox for social-networking services. These systems protect user information stored on the platform (e.g., users' profiles and social relationships). Hails [21] protects user data on the platform using language-level information flow control. Unlike πBox, none of these systems can protect private information that apps directly receive or infer from their interactions with the users.

In xBook, each user decides whether to allow a particular domain to access a given part of the user's profile. By contrast, πBox simplifies users' decision-making by color-coding the apps based on their potential for privacy violations. xBook anonymizes app statistics (with no formal privacy guarantees), while the system of Viswanath et al. uses conventional differential privacy. As we show in Section 5.2, this can lead to high relative errors when releasing rarely updated values.

Embassies [26] is somewhat similar to πBox in that it aims to secure apps through a minimal interface that allows most apps to function correctly. Unlike in πBox, app publishers are not viewed as adversaries with respect to the user data collected by the app.

Dynamic taint analysis tracks the flow of sensitive data through program binaries [10, 24, 62] and can help protect user privacy. For example, TaintDroid [18] detects (rather than prevents) privacy violations, while AppFence [25] uses data shadowing and exfiltration blocking to prevent tainted data from leaving the device. Neither system handles implicit leaks. While taint-based systems can track specific data items such as device ID, they cannot prevent the app from leaking information about the user's behavior (e.g., articles the user has read). In general, dynamic taint tracking is complementary to the guarantees provided by πBox. For example, it can be used to prevent certain data items from being declassified even via differentially private channels.

Bubbles [55] aims to capture privacy intentions by clustering data into "bubbles" based on explicit user behavior. The privacy guarantee is similar to that of πBox's sharing channel: once the user adds a friend to a bubble, this friend gains access to all data in that bubble. Bubbles is limited to apps that run on the client device only.

ObliviAd [5] and PrivAd [22] are privacy-preserving online advertising systems that aim to protect user profiles from ad brokers. ObliviAd creates a black box at the ad broker using a secure coprocessor and oblivious RAM. This black box serves ads to clients, receives reports about ad clicks and impressions from clients via a secure TLS channel, records which ads were clicked or viewed (but not who viewed an ad), and only releases these records in large batches to make it difficult to determine who saw which ad. In PrivAd, clients fetch a large set of ads that are roughly based on users' interests; more accurate targeting is done only at the client. When the client reports which ads have been shown, a trusted third party anonymizes his identity before sending the data to the ad broker. By contrast, πBox aims to provide rigorous privacy guarantees without sacrificing the ability of advertisers to obtain accurate impression counts.

PINQ [35] and Airavat [48] are centralized platforms for differentially private computations on static datasets. PDDP [9] is a distributed differential privacy system in which participants maintain their own data.

While the cloud provider is trusted in πBox, CloudVisor [61] and CryptDB [44] focus on untrusted clouds. CloudVisor hides users' data from the hypervisor using nested virtualization, CryptDB uses encryption. CLAMP [42] employs isolation and authentication mechanisms that are similar to πBox to protect private data in LAMP-like Web servers. It focuses on compromised servers rather than malicious applications.

πBox can be viewed as imposing a mandatory information flow policy on untrusted apps. Previous work on information flow control includes [12, 32, 38, 60] and hundreds of other papers.

Bring-Your-Own-Device approaches that support dual workspaces [3, 58] enable personal and corporate data to coexist on the same device while permitting only trusted apps to access the corporate data. πBox takes this idea a step further and allows untrusted apps to run on corporate data, thus realizing the idea of Bring-Your-Own-App.

7 Conclusion

πBox is a new app platform that combines support for apps' functional needs with rigorous privacy protection for their users. Our evaluation demonstrates that πBox can be used in many practical scenarios, including "bring-your-own-app" enterprise deployments where external apps operate on proprietary company data.

Acknowledgments. This work was partially supported by the NSF grants CNS-0720649, CNS-0746888, CNS-0905602, CCF-1048269, and CNS-1223396, two Google research gifts, the MURI program under AFOSR grant FA9550-08-1-0352, and grant R01 LM011028-01 from the National Library of Medicine, NIH.

References

[1] M. S. Alvim, M. E. Andrés, K. Chatzikokolakis, P. Degano, and C. Palamidessi. Differential privacy: On the trade-off between utility and information leakage. *CoRR*, abs/1103.5188, 2011.

[2] Android malware promises video while stealing contacts. http://blogs.mcafee.com/mcafee-labs/android-malware-promises-video-while-stealing-contacts.

[3] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: A virtual mobile smartphone architecture. In *SOSP*, 2011.

[4] Google App Engine. https://developers.google.com/appengine.

[5] M. Backes, A. Kate, M. Maffei, and K. Pecina. Obliviad: Provably secure and practical online behavioral advertising. In *S&P*, 2012.

[6] G. Barthe and B. Kopf. Information-theoretic bounds for differentially private mechanisms. In *CSF*, 2011.

[7] R. Bhaskar, S. Laxman, A. Smith, and A. Thakurta. Discovering frequent patterns in sensitive data. In *KDD*, 2010.

[8] J. A. Calandrino, A. Kilzer, A. Narayanan, E. W. Felten, and V. Shmatikov. "You might also like:" privacy risks of collaborative filtering. In *S&P*, 2011.

[9] R. Chen, A. Reznichenko, P. Francis, and J. Gehrke. Towards statistical queries over distributed private user data. In *NSDI*, 2012.

[10] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security*, 2004.

[11] Cloud computing security policies, procedures lacking. http://www.crn.com/news/security/224201359/cloud-computing-security-policies-procedures-lacking.htm.

[12] D. E. Denning. A lattice model of secure information flow. *Communication of the ACM*, 19(5), May 1976.

[13] I. Dinur and K. Nissim. Revealing information while preserving privacy. In *PODS*, 2003.

[14] C. Dwork. Differential privacy. In *ICALP*, 2006.

[15] C. Dwork, F. Mcsherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, 2006.

[16] C. Dwork, M. Naor, T. Pitassi, and G. N. Rothblum. Differential privacy under continual observation. In *STOC*, 2010.

[17] J. Edwards. How Microsoft's 'Do Not Track' policy is a mortal threat to ad exchanges like Facebook's. http://www.businessinsider.com/microsofts-dnt-threatens-facebook-2012-9.

[18] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[19] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *CCS 2011*.

[20] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *SOUPS*, 2012.

[21] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *OSDI*, 2012.

[22] S. Guha, B. Cheng, and P. Francis. Privad: Practical privacy in online advertising. In *NSDI*, 2011.

[23] HBase. http://hbase.apache.org.

[24] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *EuroSys*, 2006.

[25] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications. In *CCS*, 2011.

[26] J. Howell, B. Parno, and J. R. Douceur. Embassies: Radically refactoring the web. In *NSDI*, 2013.

[27] W.-M. Hu. Reducing timing channels with fuzzy time. In *S&P*, 1991.

[28] IBM bans Dropbox, Siri and rival cloud tech at work. http://www.theregister.co.uk/2012/05/25/ibm_bans_dropbox_siri.

[29] Jetty. http://www.eclipse.org/jetty.

[30] A. Korolova. Privacy violations using microtargeted ads: A case study. *Journal of Privacy and Confidentiality*, 3(1), 2011.

[31] M. Krohn and E. Tromer. Noninterference for a practical DIFC-based operating system. In *S&P*, 2009.

[32] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.

[33] B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10), Oct. 1973.

[34] App sandbox in depth. http://developer.apple.com/library/mac/#documentation/Security/Conceptual/AppSandboxDesignGuide/AppSandboxInDepth/AppSandboxInDepth.html.

[35] F. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. *Commun. ACM*, 53(9), Sept. 2010.

[36] J. Mick. Android wallpaper app stole scores of users' data, sent it to China. http://www.dailytech.com/Android+Wallpaper+App+Stole+Scores+of+Users+Data+Sent+it+to+China/article19200.htm.

[37] S. Motiee, K. Hawkey, and K. Beznosov. Do Windows users follow the principle of least privilege? Investigating User Account Control practices. In *SOUPS*, 2010.

[38] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP*, 1997.

[39] NativeClient - native code for web apps. http://code.google.com/p/nativeclient.

[40] OpenStreetMap. http://www.openstreetmap.org.

[41] OsmAnd. http://osmand.net, https://play.google.com/store/apps/details?id=net.osmand&hl=en.

[42] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen,

and A. Perrig. CLAMP: Practical prevention of large-scale data leaks. In *S&P*, 2009.

[43] iOS social apps leak contact data. http://www.informationweek.com/news/security/privacy/232600490.

[44] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

[45] Real-time bidding in the United States and Western Europe, 2010-1015. http://info.pubmatic.com/rs/pubmatic/images/IDC_Real-Time%20Bidding_US_Western%20Europe_Oct2011.pdf.

[46] A. Reznichenko, S. Guha, and P. Francis. Auctions in do-not-track compliant internet advertising. In *CCS*, 2011.

[47] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *CCS*, 2009.

[48] I. Roy, S. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for MapReduce. In *NSDI*, 2010.

[49] Saskatchewan-HTTP - seven months of HTTP logs from the University of Saskatchewan WWW Server. http://ita.ee.lbl.gov/html/contrib/Sask-HTTP.html.

[50] M. Seaborn. Plash: Tools for practical least privilege. http://plash.beasts.org.

[51] ServeStream. http://sourceforge.net/projects/servestream, https://play.google.com/store/apps/details?id=net.sourceforge.servestream&hl=en.

[52] K. Singh, S. Bhola, and W. Lee. xBook: Redesigning privacy control in social networking platforms. In *USENIX Security*, 2009.

[53] CMU Sphinx - speech recognition toolkit. http://cmusphinx.sourceforge.net.

[54] 300,000 mobile apps stealing personal data. http://www.itproportal.com/2010/07/29/300000-mobile-apps-stealing-personal-data.

[55] M. Tiwari, P. Mohan, A. Osheroff, H. Alkaff, E. Shi, E. Love, D. Song, and K. Asanović. Context-centric security. In *HotSec*, 2012.

[56] Twitter apologizes for squirreling away iPhone user data. http://www.csmonitor.com/Innovation/Horizons/2012/0216/Twitter-apologizes-for-squirreling-away-iPhone-user-data.

[57] B. Viswanath, E. Kiciman, and S. Saroiu. Keeping information safe from social networking apps. In *WOSN*, 2012.

[58] VMware Horizon Mobile. https://blogs.vmware.com/euc/2012/08/vmware-horizon-mobile-on-ios.html.

[59] 1998 World Cup website access logs. http://ita.ee.lbl.gov/html/contrib/WorldCup.html.

[60] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.

[61] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *SOSP*, 2011.

[62] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. C. Snoeren, G. M. Voelker, and S. Savage. Neon: System support for derived data management. In *VEE*, 2010.

P3: Toward Privacy-Preserving Photo Sharing

Moo-Ryong Ra Ramesh Govindan Antonio Ortega
University of Southern California

Abstract

With increasing use of mobile devices, photo sharing services are experiencing greater popularity. Aside from providing storage, photo sharing services enable bandwidth-efficient downloads to mobile devices by performing server-side image transformations (resizing, cropping). On the flip side, photo sharing services have raised privacy concerns such as leakage of photos to unauthorized viewers and the use of algorithmic recognition technologies by providers. To address these concerns, we propose a privacy-preserving photo encoding algorithm that extracts and encrypts a small, but significant, component of the photo, while preserving the remainder in a public, standards-compatible, part. These two components can be separately stored. This technique significantly reduces the accuracy of automated detection and recognition on the public part, while preserving the ability of the provider to perform server-side transformations to conserve download bandwidth usage. Our prototype privacy-preserving photo sharing system, P3, works with Facebook, and can be extended to other services as well. P3 requires no changes to existing services or mobile application software, and adds minimal photo storage overhead.

1 Introduction

With the advent of mobile devices with high-resolution on-board cameras, photo sharing has become highly popular. Users can share photos either through photo sharing services like Flickr or Picasa, or popular social networking services like Facebook or Google+. These *photo sharing service providers* (PSPs) now have a large user base, to the point where PSP photo storage subsystems have motivated interesting systems research [10].

However, this development has generated privacy concerns (Section 2). Private photos have been leaked from a prominent photo sharing site [15]. Furthermore, widespread concerns have been raised about the application of face recognition technologies in Facebook [3]. Despite these privacy threats, it is not clear that the usage of photo sharing services will diminish in the near future. This is because photo sharing services provide several useful functions that, together, make for a seamless photo browsing experience. In addition to providing photo storage, PSPs also perform several server-side image transformations (like cropping, resizing and color space conversions) designed to improve user perceived latency of photo downloads and, incidentally, bandwidth usage (an important consideration when browsing photos on a mobile device).

In this paper, we explore the design of a privacy-preserving photo sharing algorithm (and an associated system) that *ensures photo privacy without sacrificing the latency, storage, and bandwidth benefits provided by PSPs*. This paper makes two novel contributions that, to our knowledge, have not been reported in the literature (Section 6). First, the design of the P3 algorithm (Section 3), which prevents leaked photos from leaking *information*, and reduces the efficacy of automated processing (e.g., face detection, feature extraction) on photos, while still permitting a PSP to apply image transformations. It does this by splitting a photo into a public part, which contains most of the *volume* (in bytes) of the original, and a secret part which contains most of the original's *information*. Second, the design of the P3 system (Section 4), which requires no modification to the PSP infrastructure or software, and no modification to existing browsers or applications. P3 uses interposition to transparently encrypt images when they are uploaded from clients, and transparently decrypt and reconstruct images on the recipient side.

Evaluations (Section 5) on four commonly used image data sets, as well as micro-benchmarks on an implementation of P3, reveal several interesting results. Across these data sets, there exists a "sweet spot" in the parameter space that provides good privacy while at the same time preserving the storage, latency, and bandwidth benefits offered by PSPs. At this sweet spot, algorithms like edge detection, face detection, face recognition, and SIFT feature extraction are completely ineffective; *no* faces can be detected and correctly recognized from the public part, *no* correct features can be extracted, and a very small fraction of pixels defining edges are correctly estimated. P3 image encryption and decryption are fast, and it is able to reconstruct images accurately even when the PSP's image transformations are not publicly known.

P3 is proof-of-concept of, and a step towards, easily deployable privacy preserving photo storage. Adoption of this technology will be dictated by economic incen-

tives: for example, PSPs can offer privacy preserving photo storage as a premium service offered to privacy-conscious customers.

2 Background and Motivation

The focus of this paper is on PSPs like Facebook, Picasa, Flickr, and Imgur, who offer either direct *photo sharing* (e.g., Flickr, Picasa) between users or have integrated photo sharing into a social network platform (e.g., Facebook). In this section, we describe some background before motivating privacy-preserving photo sharing.

2.1 Image Standards, Compression and Scalability

Over the last two decades, several standard image formats have been developed that enable interoperability between producers and consumers of images. Perhaps not surprisingly, most of the existing PSPs like Facebook, Flickr, Picasa Web, and many websites [6, 7, 41] primarily use the most prevalent of these standards, the JPEG (Joint Photographic Experts Group) standard. In this paper, we focus on methods to preserve the privacy of JPEG images; supporting other standards such as GIF and PNG (usually used to represent computer-generated images like logos etc.) are left to future work.

Beyond standardizing an image file format, JPEG performs lossy compression of images. A JPEG encoder consists of the following sequence of steps:

Color Space Conversion and Downsampling. In this step, the raw RGB or color filter array (CFA) RGB image captured by a digital camera is mapped to a YUV color space. Typically, the two chrominance channels (U and V) are represented at lower resolution than the luminance (brightness) channel (Y) reducing the amount of pixel data to be encoded without significant impact on perceptual quality.

DCT Transformation. In the next step, the image is divided into an array of blocks, each with 8×8 pixels, and the Discrete Cosine Transform (DCT) is applied to each block, resulting in several *DCT coefficients*. The mean value of the pixels is called the DC coefficient. The remaining are called AC coefficients.

Quantization. In this step, these coefficients are quantized; this is the only step in the processing chain where information is lost. For typical natural images, information tends to be concentrated in the lower frequency coefficients (which on average have larger magnitude than higher frequency ones). For this reason, JPEG applies different quantization steps to different frequencies. The degree of quantization is user-controlled and can be varied in order to achieve the desired trade-off between quality of the reconstructed image and compression rate. We note that in practice, images shared through PSPs tend to be uploaded with high quality (and high rate) settings.

Entropy Coding. In the final step, redundancy in the quantized coefficients is removed using variable length encoding of non-zero quantized coefficients and of runs of zeros in between non-zero coefficients.

Beyond storing JPEG images, PSPs perform several kinds of transformations on images for various reasons. First, when a photo is uploaded, PSPs statically resize the image to several fixed resolutions. For example, Facebook transforms an uploaded photo into a thumbnail, a "small" image (130x130) and a "big" image (720x720). These transformations have multiple uses: they can reduce storage[1], improve photo access latency for the common case when users download either the big or the small image, and also reduce bandwidth usage (an important consideration for mobile clients). In addition, PSPs perform *dynamic* (i.e., when the image is accessed) server-side transformations; they may resize the image to fit screen resolution, and may also *crop* the image to match the view selected by the user. (We have verified, by analyzing the Facebook protocol, that it supports both of these dynamic operations). These dynamic server-side transformations enable low latency access to photos and reduce bandwidth usage. Finally, in order to reduce user-perceived latency further, Facebook also employs a special mode in the JPEG standard, called *progressive* mode. For photos stored in this mode, the server delivers the coefficients in increasing order (hence "progressive") so that the clients can start rendering the photo on the screen as soon as the first few coefficients are received, without having to receive all coefficients.

In general, these transformations *scale* images in one fashion or another, and are collectively called image scalability transformations. Image scalability is crucial for PSPs, since it helps them optimize several aspects of their operation: it reduces photo storage, which can be a significant issue for a popular social network platform [10]; it can reduce user-perceived latency, and reduce bandwidth usage, hence improving user satisfaction.

2.2 Threat Model, Goals and Assumptions

In this paper, we focus on two specific threats to privacy that result from uploading user images to PSPs. The first threat is unauthorized access to photos. A concrete instance of this threat is the practice of *fusking*, which attempts to reverse-engineer PSP photo URLs in order to access stored photos, bypassing PSP access controls. Fusking has been applied to at least one PSP (Photobucket), resulting in significant privacy leakage [15]. The

[1] We do not know if Facebook preserves the original image, but high-end mobile devices can generate photos with 4000x4000 resolution and resizing these images to a few small fixed resolutions can save space.

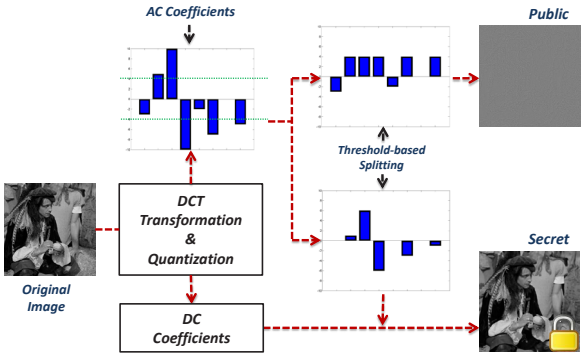


Figure 1: Privacy-Preserving Image Encoding Algorithm

second threat is posed by automatic recognition technologies, by which PSPs may be able to infer social contexts not explicitly specified by users. Facebook's deployment of face recognition technology has raised significant privacy concerns in many countries (e.g., [3]).

The goal of this paper is *to design and implement a system that enables users to ensure the privacy of their photos (with respect to the two threats listed above), while still benefiting from the image scalability optimizations provided by the PSP.*

Implicit in this statement are several constraints, which make the problem significantly challenging. The resulting system must not require any software changes at the PSP, since this is a significant barrier to deployment; an important implication of this constraint is that the image stored on the PSP must be JPEG-compliant. For a similar reason, the resulting system must also be transparent to the client. Finally, our solution must not significantly increase storage requirements at the PSP since, for large PSPs, photo storage is a concern.

We make the following assumptions about trust in the various components of the system. We assume that all local software/hardware components on clients (mobile devices, laptops etc.) are completely trustworthy, including the operating system, applications and sensors. We assume that PSPs are completely untrusted and may either by commission or omission, breach privacy in the two ways described above. Furthermore, we assume eavesdroppers may attempt to snoop on the communication between PSP and a client.

3 P3: The Algorithm

In this section, we describe the P3 algorithm for ensuring privacy of photos uploaded to PSPs. In the next section, we describe the design and implementation of a complete system for privacy-preserving photo sharing.

3.1 Overview

One possibility for preserving the privacy of photos is end-to-end encryption. *Senders*[2] may encrypt photos before uploading, and *recipients* use a shared secret key to decrypt photos on their devices. This approach cannot provide image scalability, since the photo representation is non-JPEG compliant and opaque to the PSP so it cannot perform transformations like resizing and cropping. Indeed, PSPs like Facebook reject attempts to upload fully-encrypted images.

A second approach is to leverage the JPEG image compression pipeline. Current image compression standards use a well-known *DCT dictionary* when computing the DCT coefficients. A *private* dictionary [9], known only to the sender and the authorized recipients, can be used to preserve privacy. Using the coefficients of this dictionary, it may be possible for PSPs to perform image scaling transformations. However, as currently defined, these coefficients result in a non-JPEG compliant bit-stream, so PSP-side code changes would be required in order to make this approach work.

A third strawman approach might selectively hide faces by performing face detection on an image before uploading. This would leave a JPEG-compliant image in the clear, with the hidden faces stored in a separate encrypted part. At the recipient, the image can be reconstructed by combining the two parts. However, this approach does not address our privacy goals completely: if an image is leaked from the PSP, attackers can still obtain significant information from the non-obscured parts (e.g., torsos, other objects in the background etc.).

Our approach on privacy-preserving photo sharing uses a *selective encryption* like this, but has a different design. In this approach, called P3, a photo is divided into two parts, a *public* part and a *secret* part. The public part is exposed to the PSP, while the secret part is encrypted and shared between the sender and the recipients (in a manner discussed later). Given the constraints discussed in Section 2, the public and secret parts must satisfy the following requirements:

- It must be possible to represent the public part as a JPEG-compliant image. This will allow PSPs to perform image scaling.
- However, intuitively, most of the "important" *information* in the photo must be in the secret part. This would prevent attackers from making sense of the public part of the photos even if they were able to access these photos. It would also prevent PSPs from successfully applying recognition algorithms.
- Most of the *volume* (in bytes) of the image must reside in the public part. This would permit PSP server-side

[2]We use "sender" to denote the user of a PSP who uploads images to the PSP.

image scaling to have the bandwidth and latency benefits discussed above.

- The combined size of the public and secret parts of the image must not significantly exceed the size of the original image, as discussed above.

Our P3 algorithm, which satisfies these requirements, has two components: a sender side encryption algorithm, and a recipient-side decryption algorithm.

3.2 Sender-Side Encryption

JPEG compression relies on the *sparsity* in the DCT domain of typical natural images: a few (large magnitude) coefficients provide most of the information needed to reconstruct the pixels. Moreover, as the quality of cameras on mobile devices increases, images uploaded to PSPs are typically encoded at high quality. P3 leverages both the sparsity and the high quality of these images. First, because of sparsity, most information is contained in a few coefficients, so it is sufficient to degrade a few such coefficients, in order to achieve significant reductions in quality of the public image. Second, because the quality is high, quantization of each coefficient is very fine and the least significant bits of each coefficient represent very small incremental gains in reconstruction quality. P3's encryption algorithm encode the most significant bits of (the few) significant coefficients in the secret part, leaving everything else (less important coefficients, and least significant bits of more important coefficients) in the public part. We concretize this intuition in the following design for P3 sender side encryption.

The selective encryption algorithm is, conceptually, inserted into the JPEG compression pipeline after the quantization step. At this point, the image has been converted into frequency-domain quantized DCT coefficients. While there are many possible approaches to extracting the most significant information, P3 uses a relatively simple approach. First, it extracts the DC coefficients from the image into the secret part, replacing them with zero values in the public part. The DC coefficients represent the average value of each 8x8 pixel block of the image; these coefficients usually contain enough information to represent thumbnail versions of the original image with enough visual clarity.

Second, P3 uses a threshold-based splitting algorithm in which each AC coefficient $y(i)$ whose value is above a threshold T is processed as follows:

- If $|y(i)| \leq T$, then the coefficient is represented in the public part as is, and in the secret part with a zero.
- If $|y(i)| > T$, the coefficient is replaced in the public part with T, and the secret part contains the magnitude of the difference as well as the sign.

Intuitively, this approach clips off the significant coefficients at T. T is a tunable parameter that represents the trade-off between storage/bandwidth overhead and privacy; a smaller T extracts more signal content into the secret part, but can potentially incur greater storage overhead. We explore this trade-off empirically in Section 5. Notice that both the public and secret parts are JPEG-compliant images, and, after they have been generated, can be subjected to entropy coding.

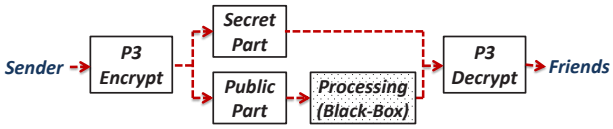


Figure 2: P3 Overall Processing Chain

Once the public and secret parts are prepared, the secret part is encrypted and, conceptually, both parts can be uploaded to the PSP (in practice, our system is designed differently, for reasons discussed in Section 4). We also defer a discussion of the encryption scheme to Section 4.

3.3 Recipient-side Decryption and Reconstruction

While the sender-side encryption algorithm is conceptually simple, the operations on the recipient-side are somewhat trickier. At the recipient, P3 must decrypt the secret part and reconstruct the original image by combining the public and secret parts. P3's selective encryption is *reversible*, in the sense that, the public and secret parts can be recombined to reconstruct the original image. This is straightforward when the public image is stored unchanged, but requires a more detailed analysis in the case when the PSP performs some processing on the public image (e.g., resizing, cropping, etc) in order to reduce storage, latency or bandwidth usage.

In order to derive how to reconstruct an image when the public image has been processed, we start by expressing the reconstruction for the unprocessed case as a series of linear operations.

Let the threshold for our splitting algorithm be denoted T. Let $\mathbf{y}$ be a block of DCT coefficients corresponding to a 8×8 pixel block in the original image. Denote $\mathbf{x_p}$ and $\mathbf{x_s}$ the corresponding DCT coefficient values assigned to the public and secret images, respectively, for the same block[3]. For example, if one of those coefficients is such that $abs(y(i)) > T$, we will have that $x_p(i) = T$ and $x_s(i) = sign(y(i))(abs(y(i)) - T)$. Since in our algorithm the sign information is encoded either in the public or in the secret part, depending on the coefficient magnitude, it is useful to explicitly consider sign information here. To do so we write $\mathbf{x_p} = \mathbf{S_p} \cdot \mathbf{a_p}$, and $\mathbf{x_s} = \mathbf{S_s} \cdot \mathbf{a_s}$, where $\mathbf{a_p}$ and $\mathbf{a_s}$ are absolute values of $\mathbf{x_p}$ and $\mathbf{x_s}$, $\mathbf{S_p}$ and $\mathbf{S_s}$ are diagonal matrices with sign information, i.e., $\mathbf{S_p} = diag(sign(\mathbf{x_p})), \mathbf{S_s} = diag(sign(\mathbf{x_s}))$. Now let $\mathbf{w}[i] = T$ if $\mathbf{S_s}[i] \neq 0$, where i is a coefficient

[3]For ease of exposition, we represent these blocks as 64x1 vectors

index, so $\mathbf{w}$ marks the positions of the above-threshold coefficients.

The key observation is that $\mathbf{x_p}$ and $\mathbf{x_s}$ *cannot be directly added* to recover $\mathbf{y}$ because the sign of a coefficient above threshold is encoded correctly *only* in the secret image. Thus, even though the public image conveys sign information for that coefficient, it might not be correct. As an example, let $y(i) < -T$, then we will have that $x_p(i) = T$ and $x_s(i) = -(abs(y(i)) - T)$, thus $x_s(i) + x_p(i) \neq y(i)$. For coefficients below threshold, $y(i)$ can be recovered trivially since $x_s(i) = 0$ and $x_p(i) = y(i)$. Note that incorrect sign in the public image occurs only for coefficients $y(i)$ above threshold, and by definition, for all those coefficients the public value is $x_p(i) = T$. Note also that removing these signs increases significantly the distortion in the public images and makes it more challenging for an attacker to approximate the original image based on only the public one.

In summary, the reconstruction can be written as a series of linear operations:

$$\mathbf{y} \quad = \quad \mathbf{S_p} \cdot \mathbf{a_p} + \mathbf{S_s} \cdot \mathbf{a_s} + \left(\mathbf{S_s} - \mathbf{S_s}^2\right) \cdot \mathbf{w} \qquad (1)$$

where the first two terms correspond to directly adding the correspondig blocks from the public and secret images, while the third term is a correction factor to account for the incorrect sign of some coefficients in the public image. This correction factor is based on the sign of the coefficients in the secret image and distinguishes three cases. If $x_s(i) = 0$ or $x_s(i) > 0$ then $y(i) = x_s(i) + x_p(i)$ (no correction), while if $x_s(i) < 0$ we have

$$y(i) = x_s(i) + x_p(i) - 2T = x_s(i) + T - 2T = x_s(i) - T.$$

Note that the operations can be very easily represented and implemented with if/then/else conditions, but the algebraic representation of (1) will be needed to determine how to operate when the public image has been subject to server-side processing. In particular, from (1), and given that the DCT is a linear operator, it becomes apparent that it would be possible to reconstruct the images in the pixel domain. That is, we could convert $\mathbf{S_p} \cdot \mathbf{a_p}$, $\mathbf{S_s} \cdot \mathbf{a_s}$ and $\left(\mathbf{S_s} - \mathbf{S_s}^2\right) \cdot \mathbf{w}$ into the pixel domain and simply add these three images pixel by pixel. Further note that the third image, the correction factor, does not depend on the public image and can be completely derived from the secret image.

We now consider the case where the PSP applies a linear operator $\mathbf{A}$ to the public part. Many interesting image transformations such as filtering, cropping[4], scaling (resizing), and overlapping can be expressed by linear operators. Thus, when the public part is requested from the PSP, $\mathbf{A} \cdot \mathbf{S_p} \cdot \mathbf{a_p}$ will be received. Then the goal is for the recipient to reconstruct $\mathbf{A} \cdot \mathbf{y}$ given the processed public image $\mathbf{A} \cdot \mathbf{S_p} \cdot \mathbf{a_p}$ and the unprocessed secret information. Based on the reconstruction formula of (1), and the linearity of $\mathbf{A}$, it is clear that the desired reconstruction can be obtained as follows

$$\mathbf{A} \cdot \mathbf{y} = \mathbf{A} \cdot \mathbf{S_p} \cdot \mathbf{a_p} + \mathbf{A} \cdot \mathbf{S_s} \cdot \mathbf{a_s} + \mathbf{A} \cdot \left(\mathbf{S_s} - \mathbf{S_s}^2\right) \cdot \mathbf{w} \qquad (2)$$

Moreover, since the DCT transform is also linear, these operations can be applied directly in the pixel domain, without needing to find a transform domain representation. As an example, if cropping is involved, it would be enough to crop the private image and the image obtained by applying an inverse DCT to $\left(\mathbf{S_s} - \mathbf{S_s}^2\right) \cdot \mathbf{w}$.

We have left an exploration of nonlinear operators to future work. It may be possible to support certain types of non-linear operations, such as pixel-wise color remapping, as found in popular apps (e.g., Instagram). If such operation can be represented as one-to-one mappings for all legitimate values[5], e.g. 0-255 RGB values, we can achieve the same level of reconstruction quality as the linear operators: at the recipient, we can reverse the mapping on the public part, combine this with the unprocessed secret part, and re-apply the color mapping on the resulting image. However, this approach can result in some loss and we have left a quantitative exploration of this loss to future work.

3.4 Algorithmic Properties of P3

Privacy Properties. By encrypting significant signal information, P3 can preserve the privacy of images by distorting them and by foiling detection and recognition algorithms (Section 5). Given only the public part, the attacker can guess the threshold T by assuming it to be the most frequent non-zero value. If this guess is correct, the attacker knows the positions of the significant coefficients, but not the range of values of these coefficients. Crucially, the sign of the coefficient is also not known. Sign information tends to be "random" in that positive and negative coefficients are almost equally likely and there is very limited correlation between signs of different coefficients, both within a block and across blocks. It can be shown that if the sign is unknown, and no prior information exists that would bias our guess, it is actually best, in terms of mean-square error (MSE), to replace the coefficient with unknown sign in the public image by 0.[6]

Finally, we observe that *proving* the privacy properties of our approach is challenging. If the public part is

[4] Cropping at 8x8 pixel boundaries is a linear operator; cropping at arbitrary boundaries can be approximated by cropping at the nearest 8x8 boundary.

[5] Often, this is the case for most color remapping operations.

[6] If an adversary sees T in the public part, replacing it with 0 will have an MSE of T^2. However, if we use any non-zero values as a guess, MSE will be at least $0.5 \times (2T)^2 = 2T^2$ because we will have a wrong sign with probability 0.5 and we know that the magnitude is at least equal to T.

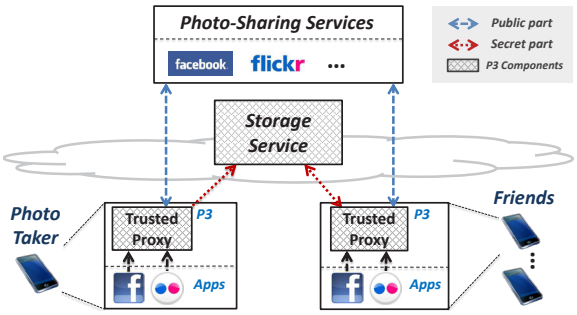


Figure 3: P3 System Architecture

leaked from the PSP, proving that no human can extract visual information from the public part would require having an accurate understanding of visual perception. Instead, we rely on metrics commonly used in the signal processing community in our evaluation (Section 5). We note that the prevailing methodology in the signal processing community for evaluating the efficacy of image and video privacy is empirical subjective evaluation using user studies, or objective evaluation using metrics [44]. In Section 5, we resort to an objective metrics-based evaluation, showing the performance of P3 on several image corpora.

Other Properties. P3 satisfies the other requirements we have discussed above. It leaves, in the clear, a JPEG-compliant image (the public part), on which the PSP can perform transformations to save storage and bandwidth. The threshold T permits trading off increased storage for increased privacy; for images whose signal content is in the DC component and a few highly-valued coefficients, the secret part can encode most of this content, while the public part contains a significant fraction of the volume of the image in bytes. As we show in our evaluation later, most images are sparse and satisfy this property. Finally, our approach of encoding the large coefficients decreases the entropy both in the public and secret parts, resulting in better compressibility and only slightly increased overhead overall relative to the unencrypted compressed image.

However, the P3 algorithm has an interesting consequence: since the secret part cannot be scaled (because, in general, the transformations that a PSP performs cannot be known a priori) and must be downloaded in its entirety, the bandwidth savings from P3 will always be lower than downloading a resized original image. The size of the secret part is determined by T: higher values of T result in smaller secret parts, but provide less privacy, a trade-off we quantify in Section 5.

4 P3: System Design

In this section, we describe the design of a system for privacy preserving photo sharing system. This system, also called P3, has two desirable properties described earlier. First, it requires no software modifications at the PSP. Second, it requires no modifications to client-side browsers or image management applications, and only requires a small footprint software installation on clients. These properties permit fairly easy deployment of privacy-preserving photo sharing.

4.1 P3 Architecture and Operation

Before designing our system, we explored the protocols used by PSPs for uploading and downloading photos. Most PSPs use HTTP or HTTPS to upload messages; we have verified this for Facebook, Picasa Web, Flickr, PhotoBucket, Smugmug, and Imageshack. This suggests a relatively simple interposition architecture, depicted in Figure 3. In this architecture, browsers and applications are configured to use a local HTTP/HTTPS *proxy* and all accesses to PSPs go through the proxy. The proxy manipulates the data stream to achieve privacy preserving photo storage, in a manner that is transparent both to the PSP and the client. In the following paragraphs, we describe the actions performed by the proxy at the sender side and at one or more recipients.

Sender-side Operation. When a sender transmits the photo taken by built-in camera, the local proxy acts as a middlebox and splits the uploaded image into a public and a secret part (as discussed in Section 3). Since the proxy resides on the client device (and hence is within the trust boundary per our assumptions, Section 2), it is reasonable to assume that the proxy can decrypt and encrypt HTTPS sessions in order to encrypt the photo.

We have not yet discussed how photos are encrypted; in our current implementation, we assume the existence of a symmetric shared key between a sender and one or more recipients. This symmetric key is assumed to be distributed out of band.

Ideally, it would have been preferable to store both the public and the secret parts on the PSP. Since the public part is a JPEG-compliant image, we explored methods to embed the secret part within the public part. The JPEG standard allows users to embed arbitrary application-specific *markers* with application-specific data in images; the standard defines 16 such markers. We attempted to use an application-specific marker to embed the secret part; unfortunately, at least 2 PSPs (Facebook and Flickr) strip all application-specific markers.

Our current design therefore stores the secret part on a cloud storage provider (in our case, Dropbox). Note that because the secret part is encrypted, we do not assume that the storage provider is trusted.

Finally, we discuss how photos are named. When a user uploads a photo to a PSP, that PSP may transform the photo in ways discussed below. Despite this, most photo-sharing services (Facebook, Picasa Web, Flickr,

Smugmug, and Imageshack[7]) assign a unique ID for all variants of the photo. This ID is returned to the client, as part of the API [21, 23], when the photo is updated.

P3's sender side proxy performs the following operations on the public and secret parts. First, it uploads the public part to the PSP either using HTTP or HTTPS (e.g., Facebook works only with HTTPS, but Flickr supports HTTP). This returns an ID, which is then used to name a file containing the secret part. This file is then uploaded to the storage provider.

Recipient-side Operation. Recipients are also configured to run a local web proxy. A client device downloads a photo from a PSP using an HTTP get request. The URL for the HTTP request contains the ID of the photo being downloaded. When the proxy sees this HTTP request, it passes the request on to the PSP, but also initiates a concurrent download of the secret part from the storage provider using the ID embedded in the URL. When both the public and secret parts have been received, the proxy performs the decryption and reconstruction procedure discussed in Section 3 and passes the resulting image to the application as the response to the HTTP get request. However, note that a secret part may be reused multiple times: for example, a user may first view a thumbnail image and then download a larger image. In these scenarios, it suffices to download the secret part once so the proxy can maintain a cache of downloaded secret parts in order to reduce bandwidth and improve latency.

There is an interesting subtlety in the photo reconstruction process. As discussed in Section 3, when the server-side transformations are known, nearly exact reconstruction is possible[8]. In our case, the precise transformations are not known, in general, to the proxy, so the problem becomes more challenging.

By uploading photos, and inspecting the results, we are able to tell, generally speaking, what kinds of transformations PSPs perform. For instance, Facebook transforms a baseline JPEG image to a progressive format and at the same time wipes out all irrelevant markers. Both Facebook and Flickr statically resize the uploaded image with different sizes; for example, Facebook generates at least three files with different resolutions, while Flickr generates a series of fixed-resolution images whose number depends on the size of the uploaded image. We cannot tell if these PSPs actually store the original images or not, and we conjecture that the resizing serves to limit storage and is also perhaps optimized for common case devices. For example, the largest resolution photos stored by Facebook is 720x720, regardless of the original resolution of the image. In addition, Facebook can dynamically resize and crop an image; the cropping geometry and the size specified for resizing are both encoded in the HTTP get URL, so the proxy is able to determine those parameters. Furthermore, by inspecting the JPEG header, we can tell some kinds of transformations that may have been performed: e.g., whether baseline image was converted to progressive or vice a versa, what sampling factors, cropping and scaling etc. were applied.

However, some other critical image processing parameters are not visible to the outside world. For example, the process of resizing an image using down sampling is often accompanied by a filtering step for antialiasing and may be followed by a sharpening step, together with a color adjustment step on the downsampled image. Not knowing which of these steps have been performed, and not knowing the parameters used in these operations, the reconstruction procedure can result in lower quality images.

To understand what transformations have been performed, we are reduced to searching the space of possible transformations for an outcome that matches the output of transformations performed by the PSP[9]. Note that this reverse engineering need only be done when a PSP re-jiggers its image transformation pipeline, so it should not be too onerous. Fortunately, for Facebook and Flickr, we were able to get reasonable reconstruction results on both systems (Section 5). These reconstruction results were obtained by exhaustively searching the parameter space with salient options based on commonly-used resizing techniques [27]. More precisely, we select several candidate settings for colorspace conversion, filtering, sharpening, enhancing, and gamma corrections, and then compare the output of these with that produced by the PSP. Our reconstruction results are presented in Section 5.

4.2 Discussion

Privacy Properties. Beyond the privacy properties of the P3 algorithm, the P3 system achieves the privacy goals outlined in Section 2. Since the proxy runs on the client for both sender and receiver, the trusted computing base for P3 includes the software and hardware device on the client. It may be possible to reduce the footprint of the trusted computing base even further using a trusted platform module [47] and trusted sensors [30], but we have deferred that to future work.

[7] PhotoBucket does not, which explains its vulnerability to fusking, as discussed earlier

[8] The only errors that can arise are due to storing the correction term in Section 3 in a lossy JPEG format that has to be decoded for processing in the pixel domain. Even if quantization is very fine, errors maybe introduced because the DCT transform is real valued and pixel values are integer, so the inverse transform of $\left(\mathbf{S_s} - \mathbf{S_s}^2\right)\mathbf{w}$ will have to be rounded to the nearest integer pixel value.

[9] This approach is clearly fragile, since the PSP can change the kinds of transformations they perform on photos. Please see the discussion below on this issue.

P3's privacy depends upon the strength of the symmetric key used to encrypt in the secret part. We assume the use of AES-based symmetric keys, distributed out of band. Furthermore, as discussed above, in P3 the storage provider cannot leak photo privacy because the secret part is encrypted. The storage provider, or for that matter the PSP, can tamper with images and hinder reconstruction; protecting against such tampering is beyond the scope of the paper. For the same reason, eavesdroppers can similarly potentially tamper with the public or the secret part, but cannot leak photo privacy.

PSP Co-operation. The P3 design we have described assumes no co-operation from the PSP. As a result, this implementation is fragile and a PSP can prevent users from using their infrastructure to store P3's public parts. For instance, they can introduce complex nonlinear transformations on images in order to foil reconstruction. They may also run simple algorithms to detect images where coefficients might have been thresholded, and refuse to store such images.

Our design is merely a proof of concept that the technology exists to transparently protect the privacy of photos, without requiring infrastructure changes or significant client-side modification. Ultimately, PSPs will need to cooperate in order for photo privacy to be possible, and this cooperation depends upon the implications of photo sharing on their respective business models.

At one extreme, if only a relatively small fraction of a PSP's user base uses P3, a PSP may choose to benevolently ignore this use (because preventing it would require commitment of resources to reprogram their infrastructure). At the other end, if PSPs see a potential loss in revenue from not being able to recognize objects/faces in photos, they may choose to react in one of two ways: shut down P3, or offer photo privacy for a fee to users. However, in this scenario, a significant number of users see value in photo privacy, so we believe that PSPs will be incentivized to offer privacy-preserving storage for a fee. In a competitive marketplace, even if one PSP were to offer privacy-preserving storage as a service, others will likely follow suit. For example, Flickr already has a "freemium" business model and can simply offer privacy preserving storage to its premium subscribers.

If a PSP were to offer privacy-preserving photo storage as a service, we believe it will have incentives to use a P3 like approach (which permits image scaling and transformations), rather than end to end encryption. With P3, a PSP can assure its users that it is only able to see the public part (reconstruction would still happen at the client), yet provide (as a service) the image transformations that can reduce user-perceived latency (which is an important consideration for retaining users of online services [10]).

Finally, with PSP co-operation, two aspects of our P3 design become simpler. First, the PSP image transformation parameters would be known, so higher quality images would result. Second, the secret part of the image could be embedded within the public part, obviating the need for a separate online storage provider.

Extensions. Extending this idea to video is feasible, but left for future work. As an initial step, it is possible to introduce the privacy preserving techniques only to the I-frames, which are coded independently using tools similar to those used in JPEG. Because other frames in a "group of pictures" are coded using an I-frame as a predictor, quality reductions in an I-frame propagate through the remaining frames. In future work, we plan to study video-specific aspects, such as how to process motion vectors or how to enable reconstruction from a processed version of a public video.

5 Evaluation

In this section, we report on an evaluation of P3. Our evaluation uses objective metrics to characterize the privacy preservation capability of P3, and it also reports, using a full-fledged implementation, on the processing overhead induced by sender and receiver side encryption.

5.1 Methodology

Metrics. Our first metric for P3 performance is the *storage overhead* imposed by selective encryption. Photo storage space is an important consideration for PSPs, and a practical scheme for privacy preserving photo storage must not incur large storage overheads. We then evaluate the efficacy of privacy preservation by measuring the performance of state-of-the-art edge and face detection algorithms, the SIFT feature extraction algorithm, and a face recognition algorithm on P3. We conclude the evaluation of privacy by discussing the efficacy of guessing attacks. We have also used PSNR to quantify privacy [43], but have omitted these results for brevity. Finally, we quantify the reconstruction performance, bandwidth savings and the processing overhead of P3.

Datasets. We evaluate P3 using four image datasets. First, as a baseline, we use the "miscellaneous" volume in the USC-SIPI image dataset [8]. This volume has 44 color and black-and-white images and contains various objects, people, scenery, and so forth, and contains many canonical images (including Lena) commonly used in the image processing community. Our second data set is from INRIA [4], and contains 1491 full color images from vacation scenes including a mountain, a river, a small town, other interesting topographies, etc. This dataset contains has greater diversity than the USC-SIPI dataset in terms of both resolutions and textures; its images vary in size up to 5 MB, while the USC-SIPI

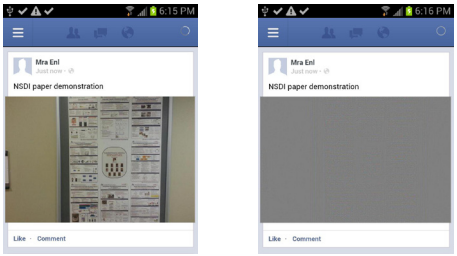
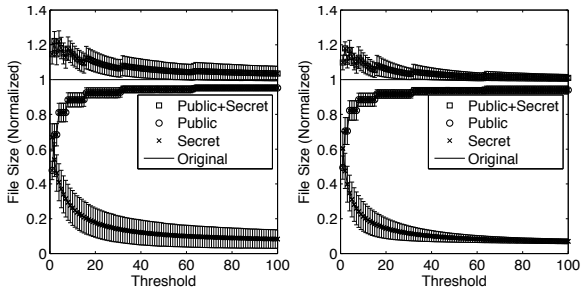
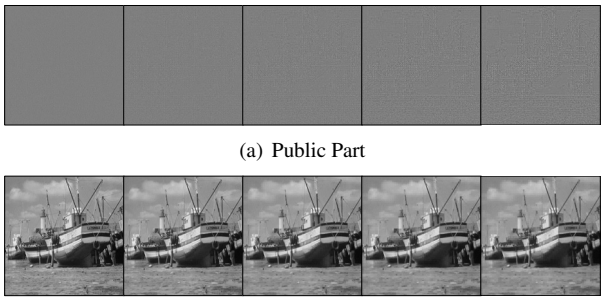


Figure 4: Screenshot(Facebook) with/without decryption



(a) USC-SIPI (b) INRIA

Figure 5: Threshold vs. Size (error bars=stdev)



(a) Public Part

(b) Secret Part

Figure 6: Baseline - Encryption Result (T: 1,5,10,15,20)

dataset's images are all under 1 MB.

We also use the Caltech face dataset [1] for our face detection experiment. This has 450 frontal color face images of about 27 unique faces depicted under different circumstances (illumination, background, facial expressions, etc.). All images contain at least one large dominant face, and zero or more additional faces. Finally, the Color FERET Database [2] is used for our face recognition experiment. This dataset is specifically designed for developing, testing, and evaluating face recognition algorithms, and contains 11,338 facial images, using 994 subjects at various angles.

Implementation. We also report results from an implementation for Facebook [20]. We chose the Android 4.x mobile operating system as our client platform, since the bandwidth limitations together with the availability of camera sensors on mobile devices motivate our work. The *mitmproxy* software tool [36] is used as a trusted man-in-the-middle proxy entity in the system. To execute a mitmproxy tool on Android, we used the *kivy/python-for-android* software [29]. Our algorithm described in Section 3 is implemented based on the code maintained by the Independent JPEG Group, version 8d [28]. We report on experiments conducted by running this prototype on Samsung Galaxy S3 smartphones.

Figure 4 shows two screenshots of a Facebook page, with two photos posted. The one on the left is the view seen by a mobile device which has our recipient-side decryption and reconstruction algorithm enabled. On the right is the same page, without that algorithm (so only the public parts of the images are visible).

5.2 Evaluation Results

In this section, we first report on the trade-off between the threshold parameter and storage size in P3. We then evaluate various privacy metrics, and conclude with an evaluation of reconstruction performance, bandwidth, and processing overhead.

5.2.1 The Threshold vs. Storage Tradoff

In P3, the threshold T is a tunable parameter that trades off storage space for privacy: at higher thresholds, fewer coefficients are in the secret part but more information is exposed in the public part. Figure 5 reports on the size of the public part (a JPEG image), the secret part (an encrypted JPEG image), and the combined size of the two parts, as a fraction of the size of the original image, for different threshold values T. One interesting feature of this figure is that, despite the differences in size and composition of the two data sets, their size *distribution as a function of thresholds is qualitatively similar*. At low thresholds (near 1), the combined image sizes exceed the original image size by about 20%, with the public and secret parts being each about 50% of the total size. While this setting provides excellent privacy, the large size of the secret part can impact bandwidth savings; recall that, in P3, the secret part has to be downloaded in its entirety even when the public part has been resized significantly. Thus, it is important to select a better operating point where the size of the secret part is smaller.

Fortunately, the shape of the curve of Figure 5 for *both datasets* suggests operating at the knee of the "secret" line (at a threshold of in the range of 15-20), where the secret part is about 20% of the original image, and the *total storage overhead is about 5-10%*. Figure 6, which depicts the public and secret parts (recall that the secret part is also a JPEG image) of a canonical image from the USC-SIPI dataset, shows that for thresholds in this range minimal visual information is present in the public part, with all of it being stored in the secret part. We include these images to give readers a visual sense of the efficacy

of P3; we conduct more detailed privacy evaluations below. This suggests that a threshold between 10-20 might provide a good balance between privacy and storage. We solidify this finding below.

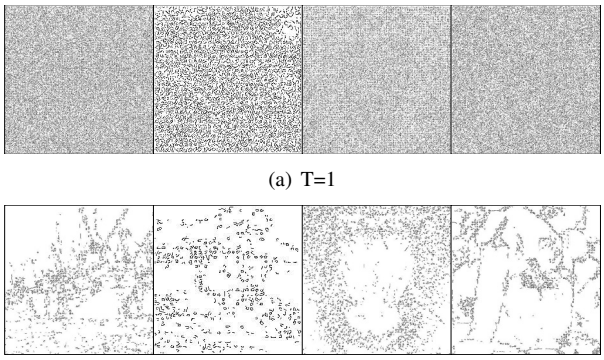
5.2.2 Privacy

In this section, we use several metrics to quantify the privacy obtained with P3. These metrics quantify the efficacy of automated algorithms on the public part; *each automated algorithm can be considered to be mounting a privacy attack on the public part.*

Edge Detection. Edge detection is an elemental processing step in many signal processing and machine vision applications, and attempts to discover discontinuities in various image characteristics. We apply the well-known Canny edge detector [14] and its implementation [24] to the public part of images in the USC-SIPI dataset, and present images with the recognized edges in Figure 8. For space reasons, we only show edges detected on the public part of 4 canonical images for a threshold of 1 and 20. The images with a threshold 20 do reveal several "features", and signal processing researchers, when told that these are canonical images from a widely used data set, can probably recognize these images. However, a layperson who has not seen the image before very likely will not be able to recognize any of the objects in the images (the interested reader can browse the USC-SIPI dataset online to find the originals). We include these images to point out that visual privacy is a highly subjective notion, and depends upon the beholder's prior experiences. If true privacy is desired, end-to-end encryption must be used. P3 provides "pretty good" privacy together with the convenience and performance offered by photo sharing services.

It is also possible to quantify the privacy offered by P3 for edge detection attacks. Figure 7(a) plots the fraction of matching pixels in the image obtained by running edge detection on the public part, and that obtained by running edge detection on the original image (the result of edge detection is an image with binary pixel values). At threshold values below 20, *barely 20% of the pixels match*; at very low thresholds, running edge detection on the public part results in a picture resembling white noise, so we believe the higher matching rate shown at low thresholds simply results from spurious matches. We conclude that, for the range of parameters we consider, P3 is very robust to edge detection.

Face Detection. Face detection algorithms detect human faces in photos, and were available as part of Facebook's face recognition API, until Facebook shut down the API [3]. To quantify the performance of face detection on P3, we use the Haar face detector from the OpenCV library [5], and apply it to the public part of images from Caltech's face dataset [1]. The efficacy of face detection, as a function of different thresholds, is shown in Figure 7(b). The y-axis represents the average number of faces detected; it is higher than 1 for the original images, because some images have more than one face. P3 *completely foils face detection* for thresholds below 20; at thresholds higher than about 35, faces are occasionally detected in some images.



(a) T=1

(b) T=20

Figure 8: Canny Edge Detection on Public Part

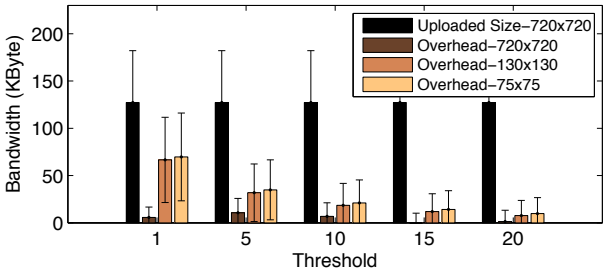
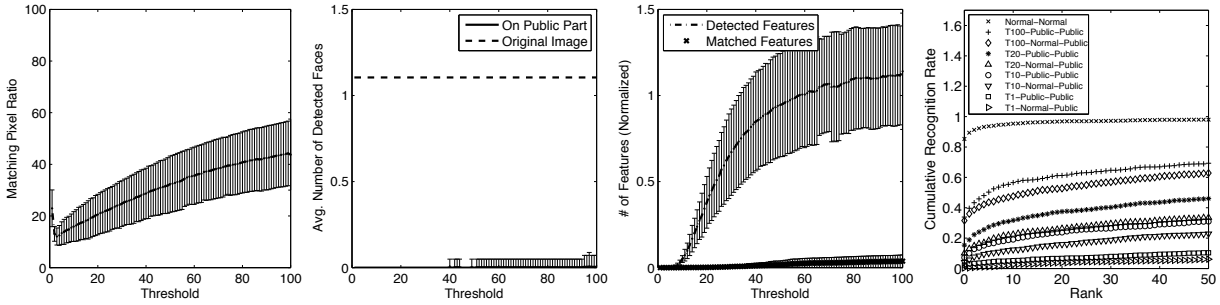


Figure 9: Bandwidth Usage Cost (INRIA)

SIFT feature extraction. SIFT [33] (or Scale-invariant Feature Transform) is a general method to detect features in images. It is used as a pre-processing step in many image detection and recognition applications from machine vision. The output of these algorithms is a set of feature vectors, each of which describes some statistically interesting aspect of the image.

We evaluate the efficacy of attacking P3 by performing SIFT feature extraction on the public part. For this, we use the implementation [32] from the designer of SIFT together with the default parameters for feature extraction and feature comparison. Figure 7(c) reports the results of running feature extraction on the USC-SIPI dataset.[10] This figure shows two lines, one of which measures the total number of features detected on the public part as a function of threshold. This shows that as the

[10]The SIFT algorithm is computationally expensive, and the INRIA data set is large, so we do not have the results for the INRIA dataset. (Recall that we need to compute for a large number of threshold values). We expect the results to be qualitatively similar.



(a) Edge Detection (b) Face Detection (c) SIFT Feature (d) Face Recognition

Figure 7: Privacy on Detection and Recognition Algorithms

threshold increases, predictably, the number of detected features increases to match the number of features detected in the original figure. More interesting is the fact that, below the threshold of 10, *no SIFT features are detected*, and below a threshold of 20, only about 25% of the features are detected.

However, this latter number is a little misleading, because we found that, in general, SIFT detects *different* feature vectors in the public part and the original image. If we count the number of features detected in the public part, which are less than a distance d (in feature space) from the nearest feature in the original image (indicating that, plausibly, SIFT may have found, in the public part, of feature in the original image), we find that this number is far smaller; up to a threshold of 35, a very small fraction of original features are discovered, and even at the threshold of 100, only about 4% of the original features have been discovered. We use the default parameter for the distance d in the SIFT implementation; changing the parameter does not change our conclusions.[11]

Face Recognition. Face recognition algorithms take an aligned and normalized face image as input and match it against a database of faces. They return the best possible answer, e.g., the closest match or an ordered list of matches, from the database. We use the Eigenface [48] algorithm and a well-known face recognition evaluation system [13] with the Color FERET database. On EigenFace, we apply two distance metrics, the Euclidean and the Mahalinobis Cosine [12], for our evaluation.

We examine two settings: *Normal-Public* setting considers the case in which training is performed on normal training images in the database and testing is executed on public parts. The *Public-Public* setting trains the database using public parts of the training images; this setting is a stronger attack on P3 than *Normal-Public*.

Figure 7(d) shows a subset of our results, based on the Mahalinobis Cosine distance metric and using the FAFB probing set in the FERET database. To quantify the recognition performance, we follow the methodology proposed by [38, 39]. In this graph, a data point at (x, y) means that y% of the time, the correct answer is contained in the top x answers returned by the EigenFace algorithm. In the absence of P3 (represented by the *Normal-Normal* line), the recognition accuracy is over 80%.

If we consider the proposed range of operating thresholds (T=1-20), the recognition rate is below 20% at rank 1. Put another way, for these thresholds, more than 80% of the time, the face recognition algorithm provides the wrong answer (a false positive). Moreover, our maximum threshold (T=20) shows about a 45% rate at rank 50, meaning that less than half the time the correct answer lies in the top 50 matches returned by the algorithm. We also examined other settings, e.g., Euclidean distance and other probing sets, and the results were qualitatively similar. These recognition rates are so low that a face recognition attack on P3 is unlikely to succeed; even if an attacker were to apply face recognition on P3, and even if the algorithm happens to be correct 20% of the time, the attacker may not be able to distinguish between a true positive and a false positive since the public image contains little visual information.

5.3 What is Lost?

P3 achieves privacy but at some cost to reconstruction accuracy, as well as bandwidth and processing overhead.

Reconstruction Accuracy. As discussed in Section 3, the reconstruction of an image for which a linear transformation has been applied should, in theory, be perfect. In practice, however, quantization effects in JPEG compression can introduce very small errors in reconstruction. Most images in the USC-SIPI dataset can be reconstructed, when the transformations are known a priori, with an average PSNR of 49.2dB. In the signal processing community, this would be considered practically

[11] Our results use a distance parameter of 0.6 from [32]; we used 0.8, the highest distance parameter that seems to be meaningful ([33], Figure 11) and the results are similar.

lossless. More interesting is the efficacy of our reconstruction of Facebook and Flickr's transformations. In Section 4, we described an exhaustive parameter search space methodology to *approximately* reverse engineer Facebook and Flickr's transformations. Our methodology is fairly successful, resulting in images with PSNR of 34.4dB for Facebook and 39.8dB for Flickr. To an untrained eye, images with such PSNR values are generally blemish-free. Thus, using P3 does not significantly degrade the accuracy of the reconstructed images.

Bandwidth usage cost. In P3, suppose a recipient downloads, from a PSP, a resized version of an uploaded image[12]. The total bandwidth usage for this download is the size of the resized public part, together with the complete secret part. Without P3, the recipient only downloads the resized version of the original image. In general, the former is larger than the latter and the difference between the two represents the bandwidth usage cost, an important consideration for usage-metered mobile data plans. This cost, as a function of the P3 threshold, is shown in Figure 9 for the INRIA dataset (the USC dataset results are similar). For thresholds in the 10-20 range, this cost is modest: 20KB or less across different resolutions (these resolutions are the ones Facebook statically resizes an uploaded image to). As an aside, the variability in bandwidth usage cost represents an opportunity: users who are more privacy conscious can choose lower thresholds at the expense of slightly higher bandwidth usage. Finally, we observe that this additional bandwidth usage can be reduced by trading off storage: a sender can upload multiple encrypted secret parts, one for each known static transformation that a PSP performs. We have not implemented this optimization.

Processing Costs. On a Galaxy S3 smartphone, for a 720x720 image (the largest resolution served by Facebook), it takes on average 152 ms to extract the public and secret parts, about 55 ms to encrypt/decrypt the secret part, and 191 ms to reconstruct the image. These costs are modest, and unlikely to impact user experience.

6 Related Work

We do not know of prior work that has attempted to address photo privacy for photo-sharing services. Our work is most closely related to work in the signal processing community on image and video privacy. Early efforts at image privacy introduced techniques like region-of-interest masking, blurring, or pixellation [17]. In these approaches, typically a face or a person in an image is represented by a blurred or pixelated version; as [17] shows, these approaches are not particularly effective against algorithmic attacks like face recognition. A subsequent generation of approaches attempted to ensure privacy for surveillance by scrambling coefficients in a manner qualitatively similar to P3's algorithm [17, 18], e.g., some of them randomly flips the sign information. However, this line of work has not explored designs under the constraints imposed by our problem, namely the need for JPEG-compliant images at PSPs to ensure storage and bandwidth benefits, and the associated requirement for relatively small secret parts.

This strand is part of a larger body of work on selective encryption in the image processing community. This research, much of it conducted in the 90s and early 2000s, was motivated by ensuring image secrecy while reducing the computation cost of encryption [35, 31]. This line of work has explored some of the techniques we use such as extracting the DC components [46] and encrypting the sign of the coefficient [45, 40], as well as techniques we have not, such as randomly permuting the coefficients [46, 42]. Relative to this body of work, P3 is novel in being a selective encryption scheme tailored towards a novel set of requirements, motivated by photo sharing services. In particular, to our knowledge, prior work has not explored selective encryption schemes which permit image reconstruction when the unencrypted part of the image has been subjected to transformations like resizing or cropping. Finally, a pending patent application by one of the co-authors [37] of this paper, includes the idea of separating an image into two parts, but does not propose the P3 algorithm, nor does it consider the reconstruction challenges described in Section 3.

Tangentially related is a body of work in the computer systems community on ensuring other forms of privacy: secure distributed storage systems [22, 34, 11], and privacy and anonymity for mobile systems [19, 25, 16]. None of these techniques directly apply to our setting.

7 Conclusions

P3 is a privacy preserving photo sharing scheme that leverages the sparsity and quality of images to store most of the information in an image in a secret part, leaving most of the volume of the image in a JPEG-compliant public part, which is uploaded to PSPs. P3's public parts have very low PSNRs and are robust to edge detection, face detection, or sift feature extraction attacks. These benefits come at minimal costs to reconstruction accuracy, bandwidth usage and processing overhead.

Acknowledgements. We would like to thank our shepherd Bryan Ford and the anonymous referees for their insightful comments. This research was sponsored in part under the U.S. National Science Foundation grant CNS-1048824. Portions of the research in this paper use the FERET database of facial images collected under the FERET program, sponsored by the DOD Counterdrug Technology Development Program Office [39, 38].

[12] In our experiments, we mimic PSP resizing using ImageMagick's convert program [26]

References

[1] Caltech computational vision group, http://www.vision.caltech.edu/html-files/archive.html.

[2] The color feret database, http://www.nist.gov/itl/iad/ig/colorferet.cfm.

[3] Facebook Shuts Down Face Recognition APIs After All, http://www.theregister.co.uk/2012/07/09/facebook_face_apis_dead/.

[4] Inria holidays dataset, http://lear.inrialpes.fr/~jegou/data.php.

[5] Open source computer vision, http://opencv.willowgarage.com/wiki/.

[6] Usage of image file formats for websites, http://w3techs.com/technologies/overview/image_format/all.

[7] Usage of JPEG for websites, http://w3techs.com/technologies/details/im-jpeg/all/all.

[8] Usc-sipi image database, http://sipi.usc.edu/database/.

[9] M. Aharon, M. Elad, and A. Bruckstein. K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation. *Signal Processing, IEEE Transactions on*, 54(11):4311–4322, Nov 2006.

[10] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: facebook's photo storage. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[11] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 31–46, New York, NY, USA, 2011. ACM.

[12] R. Beveridge, D. Bolme, M. Teixeira, and B. Draper. The csu face identification evaluation system user's guide: version 5.0. *Technical Report, Computer Science Department, Colorado State University*, 2(3), 2003.

[13] R. Beveridge and B. Draper. Evaluation of Face Recognition Algorithms, http://www.cs.colostate.edu/evalfacerec/.

[14] J. Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6):679–698, June 1986.

[15] CNN: Photobucket leaves users exposed. http://articles.cnn.com/2012-08-09/tech/tech_photobucket-privacy-breach.

[16] C. Cornelius, A. Kapadia, D. Kotz, D. Peebles, M. Shin, and N. Triandopoulos. Anonysense: privacy-aware people-centric sensing. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, MobiSys '08, pages 211–224, New York, NY, USA, 2008. ACM.

[17] F. Dufaux and T. Ebrahimi. A framework for the validation of privacy protection solutions in video surveillance. In *Multimedia and Expo (ICME), 2010 IEEE International Conference on*, pages 66–71. IEEE, 2010.

[18] T. Ebrahimi. Privacy Protection of Visual Information. In *The Tutorial in MediaSense 2012, Dublin, Ireland, 21-22 May 2012*.

[19] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[20] Facebook. http://www.facebook.com.

[21] Facebook API: Photo. http://developers.facebook.com/docs/reference/api/photo/.

[22] 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 USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–, Berkeley, CA, USA, 2010. USENIX Association.

[23] Flickr API. http://www.flickr.com/services/api/upload.api.html.

[24] B. C. Haynor. A fast edge detection implementation in c, http://code.google.com/p/fast-edge/.

[25] B. Hoh, M. Gruteser, R. Herring, J. Ban, D. Work, J.-C. Herrera, A. M. Bayen, M. Annavaram, and Q. Jacobson. Virtual trip lines for distributed privacy-preserving traffic monitoring. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, MobiSys '08, pages 15–28, New York, NY, USA, 2008. ACM.

[26] ImageMagick: Convert, Edit, Or Compose Bitmap Images. http://www.imagemagick.org/.

[27] ImageMagick Resize or Scaling. http://www.imagemagick.org/Usage/resize/.

[28] Independent JPEG Group. http://www.ijg.org/.

[29] Kivy. python-for-android, https://github.com/kivy/python-for-android.

[30] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software abstractions for trusted sensors. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, pages 365–378, New York, NY, USA, 2012. ACM.

[31] X. Liu and A. M. Eskicioglu. Selective encryption of multimedia content in distribution networks: challenges and new directions. In *Conf. Communications, Internet, and Information Technology*, pages 527–533, 2003.

[32] D. Lowe. Sift keypoint detector, http://www.cs.ubc.ca/~lowe/keypoints/.

[33] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, Nov. 2004.

[34] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud Storage with Minimal Trust. In *OSDI 2010*, Oct. 2010.

[35] A. Massoudi, F. Lefebvre, C. De Vleeschouwer, B. Macq, and J.-J. Quisquater. Overview on selective encryption of image and video: challenges and perspectives. *EURASIP J. Inf. Secur.*, 2008:5:1–5:18, Jan. 2008.

[36] mitmproxy. `http://mitmproxy.org`.

[37] A. Ortega, S. Darden, A. Vellaikal, Z. Miao, and J. Caldarola. Method and system for delivering media data, Jan. 29 2002. US Patent App. 20,060/031,558.

[38] P. Phillips, H. Moon, S. Rizvi, and P. Rauss. The feret evaluation methodology for face-recognition algorithms. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(10):1090–1104, 2000.

[39] P. Phillips, H. Wechsler, J. Huang, and P. Rauss. The feret database and evaluation procedure for face-recognition algorithms. *Image and vision computing*, 16(5):295–306, 1998.

[40] C. ping Wu and C.-C. J. Kuo. Fast Encryption Methods for Audiovisual Data Confidentiality. In *in Multimedia Systems and Applications III, ser. Proc. SPIE*, pages 284–295, 2000.

[41] R. Pingdom. New facts and figures about image format use on websites. `http://royal.pingdom.com/`.

[42] L. Qiao, K. Nahrstedt, and M.-C. Tam. Is MPEG encryption by using random list instead of zigzag order secure? In *Consumer Electronics, 1997. ISCE '97., Proceedings of 1997 IEEE International Symposium on*, pages 226 – 229, Dec 1997.

[43] M.-R. Ra, R. Govindan, and A. Ortega. "P3: Toward Privacy-Preserving Photo Sharing". Technical Report arXiv: 1302.5062, http://arxiv.org/abs/1302.5062, 2013.

[44] I. Richardson. *The H. 264 advanced video compression standard*. Wiley, 2011.

[45] C. Shi and B. K. Bhargava. A Fast MPEG Video Encryption Algorithm. In *ACM Multimedia*, pages 81–88, 1998.

[46] L. Tang. Methods for encrypting and decrypting MPEG video data efficiently. In *Proceedings of the fourth ACM international conference on Multimedia*, MULTIMEDIA '96, pages 219–229, New York, NY, USA, 1996. ACM.

[47] Trusted Computing Group. TPM Main Specification, `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`.

[48] M. Turk and A. Pentland. Eigenfaces for recognition. *J. Cognitive Neuroscience*, 3(1):71–86, Jan. 1991.

Embassies: Radically Refactoring the Web

Jon Howell, Bryan Parno, John R. Douceur, *Microsoft Research*

Abstract

Web browsers ostensibly provide strong isolation for the client-side components of web applications. Unfortunately, this isolation is weak in practice; as browsers add increasingly rich APIs to please developers, these complex interfaces bloat the trusted computing base and erode cross-app isolation boundaries.

We reenvision the web interface based on the notion of a *pico-datacenter*, the client-side version of a shared server datacenter. Mutually untrusting vendors run their code on the user's computer in low-level native code containers that communicate with the outside world only via IP. Just as in the cloud datacenter, the simple semantics makes isolation tractable, yet native code gives vendors the freedom to run any software stack. Since the datacenter model is designed to be robust to malicious tenants, it is never dangerous for the user to click a link and invite a possibly-hostile party onto the client.

1 Introduction

A defining feature of the web application model is its ostensibly strong notion of isolation. On the desktop, a user use caution when installing apps, since if an app misbehaves, the consequences are unbounded. On the web, if the user clicks on a link and doesn't like what she sees, she clicks the 'close' button, and web app isolation promises that the closed app has no lasting effect on the user's experience.

Sadly, the promise of isolation is routinely broken, and so in practice, we caution users to avoid clicking on "dangerous links". Isolation fails because the web's API, responsible for application isolation, has simultaneously pursued application richness, accreting HTTP, MIME, HTML, DOM, CSS, JavaScript, JPG, PNG, Java, Flash, Silverlight, SVG, Canvas, and more. This richness introduces so much complexity that any precise specification of the web API is virtually impossible. Yet we can't hope for correct application isolation until we can specify the API's semantics. Thus, the current web API is a battle between isolation and richness, and isolation is losing.

The same battle was fought—and lost—on the desktop. The initially-simple conventional OS evolved into a rich, complex desktop API, an unmanageable disaster of complexity. Is there hope? Or do isolation (via simple specification) and richness inevitably conflict?

There is, in fact, a context in which mutually-untrusting participants interact in near-perfect autonomy, maintaining arbitrarily strong isolation in the face of evolving complexity. On the Internet, application providers, or *vendors*, run server-side applications over which they exercise total control, from the app down to the network stack, firewall, and OS. Even when vendors are tenants of a shared datacenter, each tenant autonomously controls its software stack down to the machine code, and each tenant is accessible only via IP. The strong isolation among virtualized Infrastructure-as-a-Service datacenter tenants derives not from physical separation but from the execution interface's simplicity.

This paper extends the semantics of datacenter relationships to the client's web experience. Suspending disbelief momentarily, suppose every client had ubiquitous high-performance Internet connectivity. In such a world, exploiting datacenter semantics is easy: The client is merely a *screencast* (VNC) viewer; every app runs on its vendor's servers and streams a video of its display to the client. The client bears only a few responsibilities, primarily around providing a *trusted path*, i.e., enabling the user to select which vendor to interact with and providing user input authenticity and privacy.

We can restore reality by moving the vendors' code down to the client, with the client acting as a notional *pico-datacenter*. On the client, apps enjoy fast, reliable access to the display, but the semantics of isolation remain identical to the server model: Each vendor has autonomous control over its software stack, and each vendor interacts with other vendors (remote *and* local) only through opt-in network protocols.

The pico-datacenter abstraction offers an escape from the battle between isolation and richness, by deconflating the goals into two levels of interface. The client implements the *client execution interface* (CEI), which is dedicated to isolating applications and defines how a vendor's bag of bits is interpreted by the client. Different vendors may employ, inside their isolated containers, different *developer programming interfaces* (DPIs). Today's web API is stuck in a painful battle because it conflates these goals into a single interface [11]: The API is simultaneously a collection of rich, expressive DPI functions for app developers, and also a CEI that separates vendors. The conflated result is a poor CEI that is neither simple nor well-defined. Indeed, this conflation explains why it took a decade to prevent text coloring from leaking private information [63], and why today's web allows cross-site fetches of JPGs or JavaScript but not XML [67]. The semantics of web app isolation wind through a teetering stack of rich software layers.

We deconflate the CEI and DPI by following the pico-datacenter analogy, arriving at a concrete client architecture called Embassies.[1] We pare the web CEI down to isolated native code picoprocesses [25], IP for communication beyond the process, and minimal low-level UI primitives to support the new display responsibilities identified above.

The rich DPI, on the other hand, becomes part of the web app itself, giving developers unparalleled freedom. This proposal doesn't require Alice, a web app developer, to start coding in assembly. When she writes a geotagging site, she codes against the familiar HTML, CSS, and JavaScript DPI. But, per the datacenter model, that DPI is implemented by the WebKit library [62] that Alice's client code links against, just as her server-side code links against PHP. Because Alice chooses the library, browser incompatibilities disappear.

Suppose a buffer overflow is discovered in libpng [50], a library Alice's DPI uses to draw images. Because Alice links WebKit by reference, as soon as the WebKit developers patch the bug, her client code automatically inherits the fix. Just like when Alice fixes a bug in libphp on her server, the user needn't care about this update.

Later, Alice adds a comment forum to her application. Rendering user-generated HTML has always been risky, often leading to XSS vulnerabilities [29]. But Alice hears about WebGear, a fork of WebKit, that enhances HTML with sandboxes that solve this problem robustly. DPI libraries like WebGear can innovate just as browser vendors do today, but without imposing client browser upgrades; Alice simply changes her app's linkage.

Ultimately, independent development of alternative DPIs outpace WebGear, and Alice graduates to a .NET or GTK+ stack that is more powerful, or more secure, or more elegant. Alice chooses a feature-full new framework, while Bob sticks with WebBSD, a spartan framework renowned for robustness, for his encrypted chat app. Taking the complex, rich semantics out of the CEI gives developers *more* freedom, while making cross-vendor isolation—the primary guarantee established by the client—more robust than today's web API.

Via the pico-datacenter model, we develop a CEI with:

- a minimal native execution environment,
- a minimal notion of application identity,
- a minimal primitive for persistent state,
- an IP interface for all external app communication,
- and a minimal blit-based UI semantically equivalent the screencast (VNC) model discussed above.

Such an ambitious refactoring of the web interface is necessary to finally resolve the battle between rich DPIs and a simple, well-specified CEI. While it's difficult to prove such a radical change unequivocally superior, this paper aims to demonstrate that the goal is both realistic and valuable. It makes these contributions:

- With the pico-datacenter model, we exploit the lessons of autonomous datacenter tenancy in the client environment (§3), and argue that the collateral effects of the shift are mostly harmless (§8).
- We show a small, well-defined CEI specification (§3) that admits small implementations (§6.1) and hence suggests that correct isolation is achievable.
- With a variety of rich DPI implementations running against our CEI, we demonstrate that application richness is not compromised but enhanced (§6.2).
- We show how to replace the cross-app interactions baked into today's browser with bilateral protocols (§4), maintaining familiar functionality while obeying pico-datacenter semantics.
- We implement this refactoring (§5) and show that it can achieve plausible performance (§6.3, 6.4).

2 Trends in Prior Work

Embassies is not the first attempt to improve web app isolation and richness, and indeed prior proposals improve on one or both of these axes. However, *they do not provide true datacenter-style isolation* — they incorporate, for reasons of compatibility, part or all of the aggregate web API inside their trusted computing base (TCB).

2.1 Better Browsers for the Same API

Chrome and IE8+ both shift from a single process model to one that encapsulates each tab in a separate host OS process. This increases robustness to benign failures, but these modifications don't change the web interface—multiple apps still occupy one tab, and complex cross-app interactions still occur across tabs—hence isolation among web apps is still weak. OP's browser refactoring [20] is also constrained by the web API's complex semantics.

Given this constraint, IBOS pushes the idea of refactoring the browser quite far [55]. It realizes the idea of sites as first-class OS principals [26, 57], and containerizes renderers to improve isolation. IBOS must still include HTTP to define $\langle scheme, host, port \rangle$ web principals, and must use deep-packet inspection on HTML and MIME to partially enforce the Same-Origin Policy (SOP) [67]. IBOS cannot enforce the full SOP, such as the restriction on image fetching (§3.1.4).

The Gazelle browser [58] treats sites and browser plug-ins as principals to improve isolation, but like the above systems, it maintains the existing web interface. The follow-on Service OS project [59] extended this work to encompass desktop apps, flexible web principals [45], device access, and resource management [44].

[1] An embassy is an autonomous enclave executing the will of its home country; the host territory enables multiple embassies to operate side-by-side in isolation.

All of these systems restructure the browser to improve isolation, but they are hampered by adherence to the complex web interface, in which the isolation boundary is defined in part by images, JavaScript execution [67], and fonts [4, 63]. In contrast, the pico-datacenter model imposes a new interface that makes the isolation boundary obvious and sustainable.

2.2 Changing the Web API

Many have observed that the HTML DPI isn't the best API for all web apps. An early alternative was Java [19]: a new execution and isolation model. However, because the execution model was new, and no conventional libraries worked with it, Java's CEI had to incorporate a new batch of rich interfaces and functionality, starting with the AWT GUI library. These libraries expanded the CEI (and hence the TCB), weakening the promise of isolation. The practical need for a rich DPI combined with a non-native execution model led to CEI bloat.

Atlantis [41] replaces the web's DPI with a lower-level CEI. Its executes a high-level language, and hence practical deployment of the model faces the same constraints as Java: Either it offers a limited DPI until a massive effort ports existing libraries to the new language, or it caves in and admits rich native libraries as part of the CEI (such as its `renderGUIWidget` call).

Our pico-datacenter proposal naturally evokes Tahoma [9], which defines the CEI as a hardware-compatible virtual machine. However, the Tahoma CEI isn't minimal; it includes all of HTTP and XML to specify app launch, and full hardware virtualization is needlessly broad, including x86 intricacies such as I/O ports and APICs that are irrelevant to web apps. More importantly, apps interact locally through "bins," but Tahoma doesn't explain how to use them to replace conventional web-style interactions without expanding the CEI (cf. §4), or even how to download big applications without adding a trusted cache to the CEI (cf. §3.1.3).

Various browser plug-ins, such as Flash and Silverlight, expand the existing web API to give developers options other than HTML and JavaScript. Xax [25] and Native Client [66] introduced the idea of native code web plug-ins. NaCl's SFI-based isolation requires architecture-specific reasoning, significant changes to DPI toolchains, and runtime overhead. Xax uses OS page tables, an approach that our CEI maps naturally to.

While the technologies above improve various aspects of the web, the broad approach of unioning a new interface onto the existing web API does nothing to deconflate the web's DPI and CEI and may actually introduce to security vulnerabilities [28, 60].

Mobile app platforms, such as Android, introduce an app model competitive with the web's click-anything model. But Android's permissions are closer in spirit to the desktop's model: the device and its data are sacred; installing an app explicitly welcomes the app into that sacred domain. In practice, users incorrectly trust app stores to vouch for app fidelity [31]. Our inter-application protocols (§4) evoke Android's Intents, but Embassies communication uses IP, emphasizing that a message's local origin implies *nothing* about its authority. At a low level, Android isolation is implemented with Linux user IDs [46], a subtle isolation specification wound throughout a complex kernel.

At the architectural level, our proposal employs the principle of a native, low-level interface to execution and I/O, similar to the Exokernel [30]. The Exokernel, however, aimed to expose app-specific performance opportunities; in Embassies, the low-level interface serves to maximally enforce isolation boundaries among vendors. The Exokernel project said little about how to restore inter-app functionality in a principled fashion.

3 Embassies: A Client's Pico-Datacenter

Section 1 proposed a model in which the client becomes a pico-datacenter hosting mutually distrusting apps. This section describes a specific instantiation of that idea (Fig. 1), starting with the basic execution environment offered by the pico-datacenter (§3.1). The whole reason for running an app on the client (rather than in a real data center) is proximity to the UI; to exploit this, the pico-datacenter provides each app with a minimal pixel blitting interface (for transferring pixel arrays to the screen), and the primitives needed for app-to-app display management (§3.2).

The resulting CEI (Fig. 2) has only 30 system calls, each with very simple semantics. There are no deep recesses of functionality hiding behind ioctls, making the implementation, or *client kernel,* quite small (§6.1).

3.1 Execution Environment

In the datacenter, each vendor defines its own app down to native code. Applying the pico-datacenter metaphor, our proposed CEI defines an application as a process started from a boot block of native code, running in isolation in a native environment with access to basic, microkernel-like services such as memory, synchronization and threads. An app communicates with remote servers *and* other local apps via IP packets, and it bootstraps storage from a single simple CEI call.

3.1.1 Execution: Native Code

Our client-side pico-datacenter is inspired by the success of server-side Infrastructure-as-a-Service (IaaS) systems, wherein mutually distrusting server apps occupy a shared datacenter. Server-side developers can build apps atop their choice of standard virtual machine images, or they can fine-tune or even replace the entire OS, making it easier to port existing apps. Platform-as-a-Service

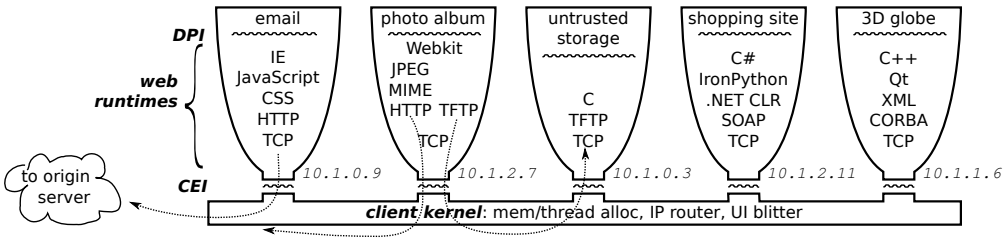


Figure 1: **The Embassies Pico-Datacenter.** *A minimal native client execution interface (**CEI**) admits a diverse set of developer programming interfaces (**DPI**s), from the web's HTML to .NET, Qt, or Gtk. Each app communicates with other servers and client apps using IP. Any protocols above IP, from TCP to HTTP to decoding a JPEG, are implemented in libraries selected by each app. Each app renders its own UI on a private framebuffer, which the **client kernel** blits to the screen (Fig. 3).*

```
allocate_memory
free_memory
process_exit

thread_create
thread_exit

futex_wait
futex_wake
get_events
get_time
set_clock_alarm

get_ifconfig
alloc_net_buffer
free_net_buffer
send_net_buffer
receive_net_buffer
get_random

get_app_secret

accept_viewport
map_canvas
update_canvas
receive_ui_event

verify_label

transfer_viewport
sublet_viewport
modify_viewport
repossess_viewport
tenant_changed

ensure_alive

endorse_me
verify_endorsement
```

Figure 2: **The Complete Embassies CEI.** *All 30 functions are non-blocking, except `futex_wait`, which can be used to wait on events that signal the completion of long-running calls.*

(PaaS) layers elegantly and efficiently on top of IaaS. Because of the simplicity of the IaaS interface, it can clearly deliver on its promise of inter-tenant isolation. Indeed, *both* Google [18] and Microsoft [43] started with PaaS but then shifted to IaaS.

By analogy, our model allows vendors to build client-side apps atop their choice of standard DPIs, including high-level languages, or they can fine-tune or replace them as desired. This reduces the pressure to bloat the CEI with new features (§2.2), since the apps can link to new feature libraries, above the level of the CEI.

While the CEI executes native-code instructions, developers obviously won't be writing code in assembly (§1). A developer writes to a high-level DPI, and the DPI implementation emits native code, including the machinery to assemble the app from a boot block.

In particular, for web apps written against the current web DPI, the functions described in this section are hidden from the developer. These functions are used by a code module called the *web runtime* (§5.2.5), which implements the web DPI.

3.1.2 Identity: Public Keys

The pico-datacenter identifies its tenants the same way entities anywhere on the open Internet are robustly identified: by associating each process with the public key of the vendor responsible for it. In other words, Embassies's principals are public keys, so an app may consist of multiple processes running different code, but Embassies will treat them all as a single principal.

Embassies identifies the principal for a process during process start. Each process starts from a self-contained, native-code boot block (§3.1.1). That boot block is signed by a private pair held by the process' principal. Before a process starts, the client kernel checks the signature, and henceforth it associates the new process with the corresponding public key; §3.2 discusses how this identity is conveyed to the user.

The CEI does not specify how the signed boot block is acquired, leaving it up to the DPIs to define and evolve suitable mechanisms — see §4.1 for an example.

The CEI also takes a data-center-based approach to handling app instances, i.e., multiple processes that belong to the same principal (i.e., public key). When a customer contacts a data-center tenant, e.g., Netflix, she contacts the vendor, rather than directly specifying a particular virtual machine running a particular binary. Similarly, with Embassies, the CEI does not specify how to contact a specific process belonging to a principal. Instead, each app vendor can choose to make all of its processes available for communication, or the vendor may choose to use one process to dispatch requests to other processes it controls.

This minimal notion of app identity contrasts with today's web, which distinguishes principals based on the protocol, host, and port used to fetch the app; thus the very specification of app identity incorporates the complexity of TCP, HTTP, HTTPS, and MIME.

Embassies's minimal definition provides a strong notion of identity, making it simple to determine when a message speaks for an application and to enable secure communication amongst apps (§3.1.4). Many awkward consequences of the web's cobbled-together definition

vanish [27]; today a vendor may own two domain names but cannot treat them as one principal, or a single domain may represent multiple entities (e.g., GeoCities or MySpace) but is treated as one principal.

However, defining and verifying app identity on an end user's client is more challenging than for a remote server, because it is not safe to download a vendor's private key to a client. For instance, Flickr uses its private key to authenticate its server, but it would never embed that key in the code it downloads to a client.

Our solution is based on the observation that, after verifying a vendor's signature on a binary, the client kernel can authoritatively state that the app *speaks for* [35] that vendor *on this machine*. The `endorse_me` call allows an app to obtain such a certification for a crypto key it generates, and other apps on the local machine can verify this with `verify_endorsement`, similar to authentication in the Nexus OS [54]. Since local apps already depend on the client kernel for correctness and security, this introduces no new dependencies.

Endorsing apps via crypto keeps the client kernel simple and makes explicit the guarantees the return value provides. It also further emphasizes the pedagogical point that each app should treat communications with local apps with as much suspicion as it would treat communications with remote apps.

3.1.3 Persistent State: Pseudorandom Keys

The current web interface specifies several local storage services as part of the CEI: an object cache, cookies, and local storage. Each service must be correct to preserve app isolation; for instance, the cache can violate an app's security or correctness if it misidentifies the origin of an object. Worse, these services have complex semantics apps cannot control; for example, the browser delivers cookies on one app's behalf when a different app makes certain requests; flaws in this design lead to Cross-Site Request Forgery (CSRF) vulnerabilities [6].

By contrast, in a shared data center, apps cannot even assume the presence of local storage, let alone complex storage APIs for caches or cookies. Instead, the app's developer uses a remote storage service, such as Amazon's S3 or Azure Storage. Even if she trusts Amazon, a sensible developer uses SSL to connect to the storage service, and a less trusting developer can use additional cryptography to avoid trusting Amazon.

Hence, following the pico-datacenter analogy, our CEI does not provide any storage services directly. Instead, apps bootstrap all of their storage needs via the `get_app_secret` call, which returns a secret specific to both the app's identity and the client machine.

The app secret is stable, so when the app restarts later, it gets the same secret. An app library can use the app secret as key material to build encrypted and authenticated storage from any untrusted external store, such as a daemon on the local client machine, a server-based cloud service, or even a peer-to-peer service. Apps use this secure storage facility to save cookies and other app-specific state.

In addition, mutually-distrusting apps can share an untrusted store that acts as a common content cache (§5.2.2); each app independently authenticates (e.g., via a MAC with the app secret as a key) the cache's content.

In both cases, replay or rollback attacks can be prevented via standard techniques [38, 48].

Our client kernel implements this interface by storing a symmetric key for a pseudorandom function (AES). It applies the function to the hash of the app's public key to generate a secret unique to the $(app, host)$ pair.

3.1.4 External Interface: IP Only

Today's web API supplies an ever-expanding set of communication primitives, including content retrieval via HTML `src` attributes, form submissions, links, JavaScript XMLHttpRequests, PostMessage, and WebSockets. Each expands the complexity of the CEI.

In contrast, our pico-datacenter follows the communication model of Internet servers: It offers only IP, with simple best-effort, non-private, non-authenticated semantics. Using IP even for messages traveling on the same machine sounds slow and counterintuitive. However, it imitates the physical constraints that guided the evolution of robust inter-server protocols. Servers communicate only by value, not by mapping shared address spaces; such decoupling leaves room to design robust protocols and select robust implementations. We can keep IP's semantics while exposing good performance by supporting bulk transfer with IPv6 jumbo frames, and by exposing a zero-copy packet interface (§5).

In practice, the client kernel assigns each app an IPv6 address and a NATed IPv4 address. The client kernel's responsibility is that of any other Internet router: best-effort delivery, with no particular guarantees on integrity or privacy.

As with any other Internet interaction, to communicate securely with other parties, an app uses cryptography. For example, the app might include a server's public key, or a public key for the root of a PKI, and then communicate with the server over SSL. The CEI does not provide cryptographic operations; the app must incorporate (e.g., via a library) any crypto code it needs. However, the CEI's `get_random` call provides a supply of secure randomness for seeding cryptographic operations, like nonce or key generation.

Communicating with Remote Servers.

In today's web, communication with remote servers is deeply complicated by the web's breathtakingly ambiguous Same Origin Policy (SOP), which refers to an ad-hoc collection of browser behaviors that attempt to selectively isolate sites from one another [67].

Locally, the SOP prevents most but not all DOM-based interactions; following the pico-datacenter metaphor, Embassies enforces a stronger, simpler policy: strictly isolate apps, with interactions only via IP.

When communicating with remote servers, the SOP primarily affects when the browser attaches cookies to an outbound request, and when a webpage can fetch content from a remote server. We discard the restrictions on cookies, since in Embassies, each app, via its DPI, governs access to its own cookies and decides when to include them in a request (§4.2). The CEI never adds ambient authority [23] to an app's communications.

The SOP's restrictions on fetching remote content aren't so easily dismissed. Since a web client may be running behind a firewall, allowing untrusted apps to freely use its network connection creates a confused-deputy vulnerability [23]. For example, an evil app on a user's web client may request content from the internal corporate payroll server, which the server allows because the request originates behind the firewall. The SOP addresses this with complicated rules such as allowing an app to retrieve an image from any site and display it, but not examine its pixels. Such rules require reasoning at a high level to know that a retrieved file *is* an image.

We observe that a much simpler policy addresses the confused-deputy threat. The threat arises from allowing untrusted apps to inherit the web client's privileged position on the network; thus, we disallow that privilege. In Embassies, every app receives, either via IT network configuration or via an explicit proxy, an IP connection logically outside any firewall. We call this "coffee-shop networking" (CSN), since apps use an IP connection semantically equivalent to a public network, e.g., in a coffee shop. An app that accesses enterprise resources can include a VPN library. To avoid asking the user to authenticate more than once, the app may choose to share its VPN connection with other enterprise-approved apps that it authenticates cryptographically (§3.1.2).

In fact, the necessary environment for CSN is emerging due to the "consumerization of IT" [47], which encourages institutions to make logically-external connections available for untrusted devices and to harden internal servers. Windows 8 grants apps an "internetClientServer" permission, a policy equivalent to CSN. [42]

We discuss the potential for resource abuse (e.g., Denial-of-Service) in §7.

Communicating with Local Applications.

In the pico-datacenter, a local app is just another server sitting on the network, and thus intra-client communication, just as app-to-server communication, is simply IP. This keeps the CEI simple and encourages defensive app design; local apps appear no different than servers because they are no more trustworthy than servers.

However, communicating with local apps differs from servers in a crucial aspect: It is reasonable to assume that server processes are available; `map.com` can send a message to `flickr.com` and reasonably expect a running process to receive it. In contrast, a web app cannot safely assume any other app is currently running on the local client.

Thus, the CEI provides the call `ensure_alive` to ensure a local process is indeed alive locally. We deliberately make the call's semantics minimal, leaving most of the work to the calling and target apps. The calling app must somehow locate the target app's binary boot block, signed by the target app's vendor, and pass it to `ensure_alive`. If no instance of the target app (as identified by the public key that signed the boot block) is yet running, the client kernel verifies the signature, starts a container for the new app, and associates the vendor's key with the container. Thereafter, the caller app can communicate with the target app by IP, for instance to pass parameters to the second app.

Note how the `ensure_alive` primitive contrasts with a conventional OS process start: no parameters, environment, handles, or library paths. A single vendor can use `ensure_alive` to create multiple processes, which may be helpful for benign fault isolation, but because each such process shares a common principal (the vendor key of §3.1.2), there is no security isolation between such processes.

3.2 UI and Display Management

The preceding subsections carve up the client machine into a fairly standard "shared datacenter"; however, a pico-datacenter is interesting because it lives near the user. Hence, unlike a traditional datacenter, we must also specify how apps access the user interface, and how the CEI handles display management. Our guiding principle is to reason about how remote, screencast apps (§1) might coordinate to manage a dumb client's UI.

User Interface. Today's web apps specify user interfaces via a complex amalgam of HTML, CSS, JavaScript, DOM, and many other standards. Our goal of a minimal CEI drives us to the leanest feasible interface: An app may accept a rectangular *viewport* region (`accept_viewport`) and map a *canvas* into its address space (`map_canvas`) – see Figure 3. This allows the client kernel to place it in a region of memory where blitting is cheap; if the viewport is resized, another call to `map_canvas` recreates a matching framebuffer. After painting pixels onto the canvas using the rendering stack it prefers, the app asks the UI (via `update_canvas`) to blit the pixels onto the visible part of the app's viewport. When the user's input focus is in the viewport, the client kernel delivers mouse and keystroke events to the app (`receive_ui_event`).

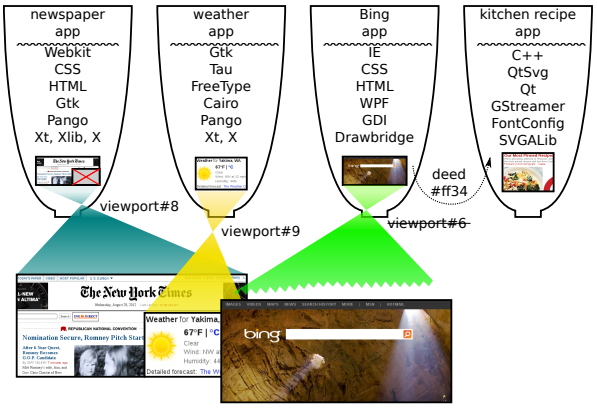


Figure 3: **UI Management.** *`Sublet_viewport` lets the newspaper nest the weather app's display inside its region. On the right, the user clicked a link on the Bing app, which used `transfer_viewport` to convert its viewport (access to the screen) into a* ***deed*** *(a secret capability), and sent the deed in a message to the kitchen app. The kitchen app will use `accept_viewport` to redeem the deed for its own viewport.*

As with the choice of native code, this refactors rich UI features into the apps, simplifying the CEI while enabling virtually any UI a DPI-developer can imagine (we discuss GPUs in §8). Indeed, because Embassies executes native code, we can employ a variety of mature UI stacks (§5.2) as DPI-supported UIs for web apps.

The client kernel labels app windows with the app's identity, so the user can select a window and know which app he is communicating with. The CEI does not use cryptographic keys directly as labels, because such keys are difficult for users to interpret. Instead, the CEI maps keys to hierarchical DNS-style labels (e.g., `bing.com`), based on and compatible with the DNSSEC PKI[2]. Before an app can accept a viewport (and hence appear on screen), the app must gather a certificate chain authenticating its label and call `verify_label`.

Naming, labeling, and visual ambiguity are hard problems; users manage to ignore most cues [52]. Our client kernel provides the minimal facility described above to address this problem, consistent with the best known methods [16, 53, 65], but we recognize that progress on this problem [10] may require CEI evolution.

Display Management. Much of today's browser functionality, such as linking, embedding, navigation, history, and tabs, are basically mechanisms for display management. To adhere to the remote screencasting abstraction (§1), we designed a viewport-management interface with capability semantics. This interface has five calls and primitive semantics; the rich browser-like functionality is built up by apps themselves (§4).

[2]Experience with SSL/TLS illustrates that deploying a large-scale PKI is challenging. Security is undermined by hundreds of certificate authorities baked into common browsers. Thus, we choose a DNSSEC-style PKI with few trust anchors and scoped naming authority.

Our CEI supports the transfer of a viewport from one app to another via `transfer_viewport`, which accepts a viewport and returns a *deed*, a secret capability that can be passed to another app via a network message. The receiving app can call `accept_viewport` to redeem the deed for a viewport it can draw in. Transforming a viewport into a deed destroys the viewport, and accepting a deed into a viewport destroys the deed; thus only one app has access to a viewport at a time.

Rather than transfer an entire viewport, an app may wish to delegate control over a rectangular sub-region of its viewport via `sublet_viewport`. This creates a deed that can be passed to another app. It also yields a handle to the sublet region, with which the parent app can resize or move the region via `modify_viewport`, or revoke it with `repossess_viewport`.

To allow communication (e.g., changes in viewport size) between the app that sublets a viewport (the *landlord*) and the app that accepts it (the *tenant*), our CEI provides each landlord-tenant pair with a fresh symmetric key that can be used to authenticate and optionally encrypt viewport-related communication. Since the key provides secrecy, integrity, and authenticity, apps may use anonymous communication mechanisms (e.g., anonymous broadcast from a random IP address) to better protect the user's privacy.

4 Refactoring Browser Interactions

§3 introduced a CEI with minimal support for hosting pico-datacenter apps and enabling them to share the UI. This section shows how we can build up equivalent functionality *inside the apps* to restore the rich cross-app interactions familiar in the classic browser. Less browser-specific interactions, such as copy-and-paste, can be handled via techniques from related work (e.g., [51]).

Rather than bake these rich interactions into the client, each interaction is reconstructed as a bilateral protocol between cooperating apps. This refactoring gives application vendors the autonomy to make security/functionality tradeoffs, for example by choosing a more robust implementation of a given protocol, implementing only a subset of it, or even refusing it altogether.

More importantly, refactoring interactions as protocols clarifies the underlying semantics, whereas in today's web, complex feature interactions lead to surprising security implications. For example, refactoring provides new perspective on Cross-Site Request Forgery (CSRF) (§4.2) and policies for visited-link coloring (§4.5).

4.1 Linking

When a classic web app includes a link to another app, it is prepared to transfer control of its screen real estate in response to the user's click. In the current web API, the hyperlink is a high-level function, bundling name reso-

lution, app fetch, app start, app window labeling, parameter passing, cookie transmission, and screen real-estate transfer into a single browser feature. In contrast, the pico-datacenter model partitions these tasks mostly between the app that contains the link and the app being linked to; the client kernel provides minimal support.

Consider `caller.net`, an Embassies app written in a classic HTML DPI, containing a hyperlink:

```
<a href="target.org/foo?x=5&y=10">
```

When a user clicks the link, the caller app identifies and contacts the target app. First, it translates `target.org` into a strong identity, perhaps by resolving it, via DNS or some stronger PKI, into a public key for the target app (§3.1.2) — §8 discusses legacy servers. Second, it contacts a local instance of the target app via local broadcast.

Since the target app may not be running locally, the caller uses `ensure_alive` (§3.1.4) to ensure that the target app has a presence on the client (in the local pico-datacenter). This requires caller.net to fetch a signed boot block matching the web runtime's ISA; it finds it as it found `target.org`'s public key. Target.org's tiny bootstrap executable retrieves and verifies the rest of its code and data, by its own means. Once `target.org`'s web runtime calls `verify_label` (§3.2), the vendor has a presence on the client.

From its client presence, `target.org` responds to caller.net's broadcast via unicast IP. The two web runtimes have their public keys endorsed by the client kernel (§3.1.2), and use them to create a secure communication channel. Caller.net's web runtime then transforms its viewport into a deed (§3.2), and sends a message to `target.org` containing the deed and the entry point parameter `/foo?x=5&y=10`. If `target.org` wishes to pass the request to its server, it does so itself (§4.2); the client kernel has no notion of HTTP. If `target.org` wishes to include a client-stored cookie, it fetches and forwards its own cookies (§3.1.3); the client kernel has no notion of HTTP cookies.

While the above process may sound heavyweight, much of it is simply a refactoring of the work done today by the browser. Furthermore, our results (§6.3) show that the overhead of app start is quite reasonable.

4.2 Cross-Domain Communication

Today's web offers many communication mechanisms, such as XMLHttpRequest, script and image inclusion, PostMessage, and third-party cookies. Refactoring them into explicit app-implemented protocols is easy.

XMLHttpRequest and HTML `script` and `image` tags use app libraries that employ TCP, HTTP, and XML libraries to reproduce standard functionality internal to the app, relying on the CEI only for IP (§3.1.4). The simplicity stems from Embassies's handling of confused-deputy problems at the IP level (§3.1.4).

PostMessage lets one local client app send messages to another. In Embassies, these messages simply become IP packets, optionally protected cryptographically.

Automatic HTML cookie semantics mixed with imperative code lead to cross-site scripting vulnerabilities; the HttpOnly attribute attempts to curtail the complexity enough to mitigate the threat [5]. In Embassies, an app can only manipulate a cookie belonging a separate vendor via an explicit IP request to the cookie's owner. The owner enforces policies on which cookies are exposed and to whom.

This refactoring reveals how CSRF threats can now be addressed by individual vendors. CSRF occurs when a malicious app dupes the browser into sending a request to a valuable app's server that's indistinguishable from a legitimate request: It looks like the user submitted a form, and it contains the valuable app's cookies. In the refactored relationship, it is straightforward for the valuable app to implement separate mechanisms for its user interactions versus its invocations from other apps.

4.3 Embedding

Visually embedding another app, such as in an `iframe`, is just like navigation, except the landlord uses `sublet_viewport` rather than `transfer_viewport`. When a sublet viewport is transferred to another app, three parties cooperate in the transfer: the old tenant, the new tenant, and the landlord. At the conclusion of the transfer, the new tenant but not the old tenant has access to the viewport, and the new tenant can communicate with the landlord without revealing its identity. The parties achieve this with a three-way protocol that performs an atomic transfer. A failed party can violate liveness, but the landlord can recover after a timeout with `repossess_viewport`.

4.4 Favorites

Classic browsers allow the user to bookmark favorite pages. This interaction becomes a protocol in Embassies: One client app acts as the user's bookmark repository. A user gesture tells an app to send a bookmark to the repository, consisting of the app's identity and an opaque entry-point parameter the app can use to reconstruct the user's state. This refactoring makes it clear that the repository gets to know which vendors the user has explicitly bookmarked, and nothing more.

4.5 Navigation Threading and History

A classic web browser tracks the user's history, enabling different views of the link graph the user traversed: the *back* button walks a path in the graph, *history* records the graph's nodes (i.e., sites the user visited), and *link coloring* displays the nodes via the current app's outbound links.

One could implement these functions in an Embassies ecosystem by declaring a trusted repository app, and adding to the linking protocol (§4.1) a step that submits a "bookmark" for the linked page to the repository.

Such a refactoring indicates that the repository is entrusted with quite a trove of private data. Furthermore, implementing link coloring reveals the repository's knowledge to every app. One could band-aid the damage by having the repository render links as embedded displays (§4.3) on behalf of apps, to avoid revealing the node graph to adversarial apps. This is essentially how the classic browser, which acts a trusted history repository, protects user privacy. Achieving privacy has been a long, complex battle [4]. In Embassies, such a relationship is at least well-defined.

However, we find the relationship too promiscuous. Instead, we deliberately abandon global history. For link coloring, we accept downgraded behavior, leaving individual applications to record their own outgoing clicks. For example, Bing can remember which links you have clicked on *from Bing*, and color such links purple. If you've arrived at `embarrassing.com` via some other path, but never from Bing, then the link to that site remains blue on Bing's results page. This provides weaker semantics than the classic web, coloring links as edges rather than nodes, but has simple privacy implications.

The back button requires each app only to know its local neighborhood of the graph. An app can provide internal navigation itself. To span apps, the linking protocol (§4.1) is extended to carry an app identity and an opaque blob, a "bookmark" for the reverse edge. When the user backs out of the target app, the target invokes the bookmark with the linking protocol to replace its display with the prior app. This allows an app to cause the back button to go to unexpected sites, break, or vanish entirely. In the classic web, the complexity of redirects and automatic navigation can cause similar mischief, rendering the browser's back button similarly problematic.

This scheme reveals the identity of the caller app to the target app, just as Referrer headers do today. The alternatives are to have a trusted, centralized store of the navigation graph (the classic browser's behavior, an approach we dislike), or to let apps create anonymous proxy identities to hide their identity from those they link to.

4.6 Window Management and Tabs

Managing overlapping windows or tabs is achieved using the same primitives that manage sublet viewports (§3.2). Thus an ordinary application, typically the first one Embassies starts, provides window resizing handles and tabs, treating the enclosed content as embedded iframes (§4.3). As with any such UI relationship in Embassies, the window manager cannot violate the privacy or integrity of the apps whose windows it manages.

The landlord controls the z-order of its tenants (presently unimplemented). The client kernel provides no support for transparency; if separate apps wish to implement it, they must expose their pixels to some app they trust to implement the blending.

5 Implementation

To evaluate the minimality and simplicity of the CEI, we implement three instantiations (§5.1). To evaluate the richness offered to developers, we port three full DPIs to Embassies (§5.2). All the code is available [1].

5.1 CEI

We have built a complete CEI implementation for Linux and a nearly complete one for the L4 microkernel [24]. For debugging purposes, we built, but omit for space, a complete non-isolating Linux implementation.

5.1.1 The Linux KVM Monitor

The measurements in §6 all run on our `linux_kvm` monitor, which relies on Linux KVM [32] to provide a virtual CPU for each app. For memory, the client kernel allocates a large contiguous block of virtual memory, and gives pieces of it to the app in response to memory requests. The client kernel performs thread scheduling, and it maintains a table of futex queues to block app threads performing `futex_wait`. It also directly implements the clock, timer, and crypto primitives.

A single central coordination process manages a connection to an X display, our UI mechanism. It also implements a logical IP subnet for routing packets between apps and to the Internet. Each app communicates with the coordinator using sockets. To connect to the Internet, the coordinator injects and intercepts packets at the IP layer using `tun`. To provide NAT, it employs the iptables functionality built into the Linux IP router. When a client is behind a firewall, it routes packets over an IP tunnel to a CSN proxy. For performance when moving large data between apps, it provides a zero-copy path for IPv6 jumbo frames, using shared memory.

5.1.2 The L4/Genode Monitor

We have also implemented the CEI on an L4::Pistachio microkernel [24], building on the Genode OS [14, 17] framework's memory allocation, RPC abstractions, and Nitpicker UI [15]. It runs all of the rich-DPI applications the Linux KVM monitor does.

5.1.3 Alternatives

While the linux_kvm monitor depends on hardware virtualization, the CEI doesn't require it. It supports any computer with an MMU [25], perhaps using OS mechanisms like seccomp [36] or PTRACE_SYSEMU.

5.2 DPIs

We have linked three full DPIs against Embassies: classic web, Gnome/Gtk, and KDE/Qt. The classic web

DPI is built from a Webkit-based [62] browser, Midori [56], which is itself built on Gtk libraries. The KDE/Qt toolkit is almost entirely distinct, but it shares its bottom layers (X, libc) with Gtk. In addition, we built a minimal DPI (§5.2.1) that runs native C code and accesses CEI facilities directly. Each DPI is a stack of software that talks to the CEI at the bottom layer.

5.2.1 POSIX Emulation

Embassies's POSIX emulation layer (EPE) lies at the bottom of each DPI we implemented. It supports the POSIX-facing libc, which in turn supports Gtk and Qt. For instance, libc implements its malloc function by calling `brk` or `mmap`, and EPE converts these into an `allocate_memory` call to our CEI.

Because POSIX identifies system resources via the filesystem namespace, EPE includes a virtual in-process filesystem (VFS) implementation, with several underlying filesystems. Implementing facilities as VFSs is often easier than modifying app logic in higher layers [25].

5.2.2 Virtual Filesystems

EPE includes a read-only filesystem that holds an image of the applications' executable and data files. EPE also contains entry-point code, which maps a copy of the dynamic loader `ld` and calls it with the path to the app executable in the read-only filesystem.

This read-only filesystem accesses data from a storage service (§3.1.3) via an FTP-like protocol. Files are identified by their hash values, which are computed using Merkle trees [40] to facilitate content-based block sharing with other apps. If the service doesn't have a requested block, the read-only filesystem contacts the app's origin server. Fetching files incurs costly round trips, so the read-only filesystem initially prefetches a tar-file of the app's startup files. Requests that fail in the tar-file fall through to individual cache requests.

To store an app's temporary files, EPE provides a RAM-disk VFS. For intra-app communication, EPE provides access to pipes and sockets via another VFS. EPE translates app reads from `/dev/random` into `get_random` CEI calls. Reads from `/proc` are partially emulated within EPE, e.g., to provide the stack layout to garbage-collection libraries. A VFS provides a filesystem for securely storing persistent data (§3.1.3), e.g., cookies; these employ a local storage service. Another VFS provides access to a server-side store.

5.2.3 Xvnc

All our DPIs are currently based on X graphics. Our implementation satisfies X requests via a modified Xvnc library. Xvnc speaks the X protocol at the top and the VNC remote-frame-buffer protocol at the bottom. We replace the bottom with code that uses our CEI's viewport/canvas instead. This modified about 350 SLoC.

5.2.4 Gtk and Qt

Once these layers are in place, getting a much richer toolkit in place is surprisingly straightforward, even though these toolkits consist of 50–100 libraries. Some Gnome-based applications were insistent that a Dbus object broker be present; we satisfy them by simply spinning one up within the app. Other apps, such as Gimp, draw numerous toolboxes. We load a twm window manager alongside Gimp to enable the user to manipulate the toolboxes on a single Embassies viewport.

5.2.5 libwebkit and Midori

For our HTML DPI, we started with Midori [56], a browser based on the libwebkit HTML DOM implementation [62]. Midori and Webkit are in turn Gtk apps, so most of their requirements are satisfied by the techniques above. We implemented a tab manager (§4.6) and inserted hooks in Webkit's link, GET, and iframe mechanisms to connect them to the linking (§4.1), navigation (§4.5), and embedding (§4.3) protocols. For example, in the link case, the hook retrieves the tenant viewport from Xvnc, converts it into a deed, and forwards it to the destination app. We have not yet implemented window management, favorites, or history management, though these should be straightforward, since window management is a subset of tab management, and favorites and history are handled by normal apps.

5.2.6 Alternative DPIs

Drawbridge ports Windows and .NET to a "picoprocess" interface close to our CEI, making it a good candidate for a web DPI [49].

5.3 Architectures

We have only implemented an x86-32 variant of the CEI. Nothing in the CEI depends on the ISA; other architectures would be straightforward. The x86 CEI variant inherits an ISA quirk: all popular x86 software frameworks abuse an x86 segment register as a thread-local store pointer to reduce pressure on the paltry x86 register file. We support this by adding a `x86_set_segment` call to the x86 CEI variant. The call has trivial semantics and no security impact; supporting it lets most library binaries run unmodified, greatly easing porting effort.

6 Evaluation

This evaluation answers four questions: Does the CEI achieve its goal of minimality (§6.1)? Does it support diverse, rich DPIs (§6.2)? We shift the burden for application bootstrapping onto apps themselves; how big is the performance cost (§6.3)? When each app brings its own DPI, is the memory burden acceptable (§6.4)?

We test with an HP z420 workstation with a four-core, 3.6GHz Intel Xeon E5-1620 CPU and 4GB of RAM.

Client Kernel	SLoC	Underlying TCB
linux_kvm	28,138	Linux (millions)
linux_dbg	21,445	Linux (millions)
bare_iron	16,714	Genode, L4 (~70K)
Firefox	4,561,642	Linux (millions)
Chrome	6,722,375	Linux (millions)

Figure 4: **TCB.** *Unlike today's web API, the Embassies CEI admits modest implementations.*

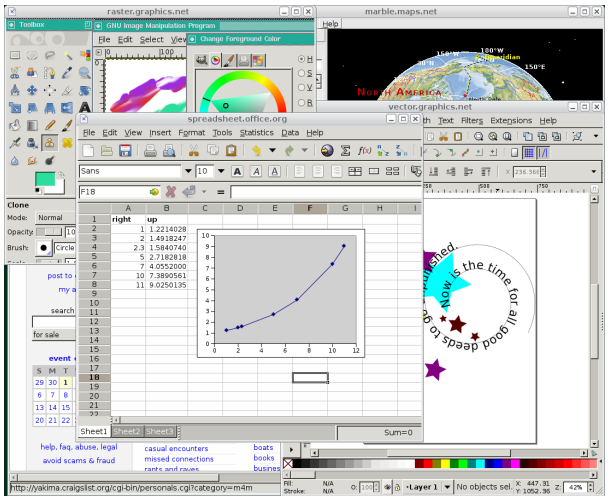


Figure 5: **Diverse DPIs.** *Native code as CEI enables diverse DPIs. This screenshot shows apps Craigslist (Webkit/HTML), Gimp (Gtk), Marble (KDE/Qt), Inkscape (Gtk), and Gnumeric (Gtk) running on the Embassies CEI. Not shown are Abiword (Gtk), Gnucash (Gtk), or Hyperoid (EPE).*

6.1 Minimality/Simplicity of the CEI

CEI minimality both improves isolation by reducing TCB size, and leaves richness up to the app's libraries. Figure 4 counts the client-kernel code sizes [64], which represents the amount of code all apps must trust. Each CEI implementation depends on some underlying OS. Although Linux is huge, CEI safety depends only on a subset of its semantics, memory management and the `kvm` driver. Likewise, the display uses X, but only pixel rectangles, not X's security model. The L4 implementation further supports the hypothesis that the Embassies CEI can be implemented with relatively little code.

Any application running *on* the CEI may include millions of lines of code, but the vendor controls *which* lines, and none of this code increases the TCB of any other app.

6.2 Diversity of DPIs

We have demonstrated half a dozen applications running on three major DPIs—Gtk, Qt, and Webkit—comprising 143 MB of binary in 200 libraries (Fig. 5).

6.3 Performance

We consider it worthwhile to spend some performance for a richer, more secure web. How much performance are we spending?

CPU Overhead. We ran a subset of the SunSpider JavaScript benchmark [61] on both Linux and Embassies. We also ran Gimp image rotations as a native macrobenchmark. Unsurprisingly, in both cases the difference is negligible: results are within 2% with standard deviations of 1%. These results confirm that a well-designed, low-level CEI need not add any additional CPU overhead to such computations.

Communication. To evaluate the overhead of IP communication between local apps, we measured the time Midori takes to fetch its cookies from an untrusted store (§3.1.3). This involves not only IP latency, but the cryptographic overhead of decrypting and verifying the integrity of the data. Nonetheless, we find that Midori can read or write a cookie in under a millisecond; refactoring interactions into protocols adds negligible overhead.

As discussed below, we use zero-copy data transfers and caching to reduce the overhead of transferring large amounts of data (e.g., DPI images) between apps.

App Start. The most significant impact of our refactoring is that, rather than intimately sharing a monolithic browser's heap, each app bootstraps its own DPI layers. How much does this increase the latency between when a user clicks a link and when the app launches?

The very first time the client ever encounters a new DPI, she must, of course, download it, just as she would if she selected a new browser. Subsequently, the DPI's files can be served rapidly out of a local, untrusted cache (E-Hot in Figure 6). Indeed, clever caches will likely preload popular DPIs to avoid even the first-time download. In a "patched" start (E-Patch, Fig. 6), the app's image is absent from the cache, but another app based on a similar DPI is present, and the Merkle tree reveals that only a delta is needed (§5.2.2). Thus, deviation from popular DPIs will result in an initial app load time proportional to the amount of deviation. One reason a vendor might deviate from a popular DPI is to fix a broken library. For example, libpng patched an overflow vulnerability in February 2012 [50]. In this case, the "patched" Midori is 76MB but differs from the cached Midori only by the 0.5MB repaired libpng library. Once the delta has been fetched, subsequent fetches by any other vendor using the patched libpng also hits the cache.

To reduce bootstrap time, we start each app from a tar file, so the entire image is transferred from the untrusted cache in one packet (§5.2.2), reducing overhead and enabling zero-copy optimizations. The first time an app runs, its loader verifies the hash (SHA-1) of the tar file; to save time on future loads, the app uses its platform-specific secret key (§3.1.3) to MAC the tar file and stores the MAC value in untrusted storage. MACs such as VMAC [33] can be verified faster than a hash.

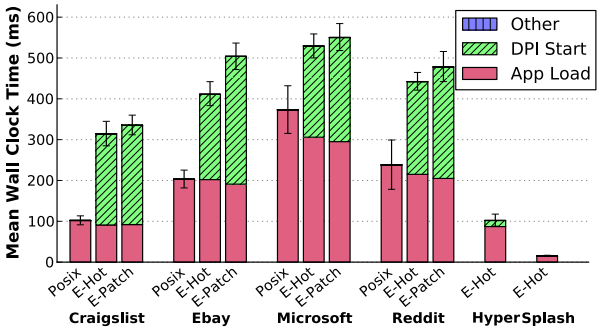


Figure 6: **App Startup Latency.** *Four web apps, a native game (Hyperoid), and a splash screen. For the web apps "App Load" is the time to fetch and render the HTML content. Error bars show standard deviations of total time over 10 runs.*

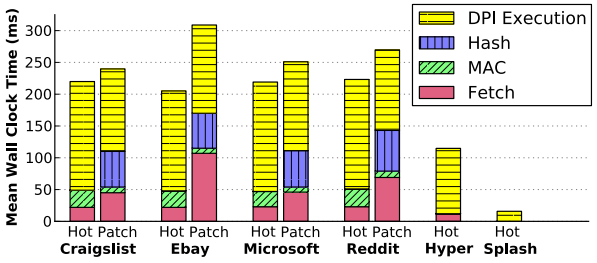


Figure 7: **DPI Start Breakdown.** *Fetching the DPI from the cache costs more in the warm case, due to fetching upstream blocks and the need to hash rather than MAC for integrity. Mean of 10 trials.*

Figure 6 assumes zero network delay to avoid burying Embassies's overheads in high network latencies. Our untrusted cache only supports UDP, incurring many RTTs hidden by this zero-delay assumption, but in deployment, it would pipeline blocks with TCP, incurring RTTs typical of HTTP transfers.

We load a set of popular websites in Midori on Linux, which takes 102–373 ms. In contrast, a hot start on Embassies takes 314–529 ms, and a patched start takes 335–551 ms. Unsurprisingly, the app load (i.e., web page fetch and render) step is similar in both cases. Embassies's overhead comes primarily from the need to fetch, verify, and boot the Midori DPI. Most of that time (Fig.7) comes from starting Midori from scratch, which even on Linux requires 130 ms ($\sigma = 7$). This is unsurprising, since Midori app starts are assumed rare, and hence unoptimized. This overhead could be mitigated by checkpointing to avoid library relocation [12, 39], by applying Midori-specific tuning (e.g., not loading every available font on startup), or by displaying a splash screen until the app achieves interactivity. Figure 6 shows that Embassies can display such a splash screen (1.5MB) in 15 ms ($\sigma = 1$). As an example of optimized start time, we ported a game, Hyperoid, to Embassies. It starts in 102 ms ($\sigma = 15$) when cached.

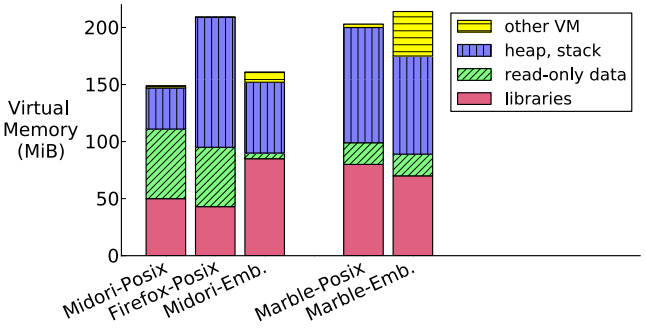


Figure 8: **Memory.** *Embassies DPI implementations consume virtual memory comparable to their POSIX progenitors.*

These costs are within the ballpark of a page load, but further improvements are possible. A hot app can remain resident to avoid a start altogether. The tar file is captured at file granularity, but many files are barely touched; page granularity would reduce the 76MB image to 33MB.

In summary, while the 177–300 ms overhead of our prototype is a non-trivial delay, there are plenty of opportunities to improve it; our refactoring makes those opportunities accessible to vendors. Overall, we are glad to exchange the challenges of security and app richness for the ordinary task of systems performance tweaking.

6.4 Memory Usage

If every vendor's application loads its own copy of a DPI implementation, will memory usage be overwhelming? Prior work shows that this style of statically linked code need not cost significantly more memory than traditional shared code implementations [8, 22].

Figure 8 contrasts virtual memory usage of POSIX implementations with those in Embassies. Since it incorporates the Xvnc rasterizer and other libraries, Midori in Embassies uses 12MB (8%) more virtual memory than its POSIX equivalent. Another DPI instance, Marble running on Qt, shows similar growth, 11MB (5%).

In a conventional browser, one instance of the browser serves many applications, amortizing fixed costs of both libraries and some heap structures. The zero-copy IP router in Embassies affords the same opportunity for libraries—the untrusted cache could send the same payload to multiple applications—but our prototype does not yet implement copy-on-write. With regard to the heap, more modern browsers (IE9 and Chrome) launch one process-per-tab, creating more heaps; in Embassies, process-per-app incurs additional heap costs.

7 Security Analysis

Embassies improves security by specifying such a small, simple CEI that implementations thereof stand a reasonable chance of truly fulfilling the web's promise of app isolation. The client kernel's small TCB (§6.1) means that the amount of code all apps must trust is

tiny, and hence each vendor can independently choose the right tradeoff between complex functionality and security. A gaming app can use a rich, full-featured DPI, while a banking app may choose a conservative DPI enhanced with the latest security protections. One app's insecurity never undermines the security of other apps. Finally, since the pico-datacenter model deconflates the CEI from the DPI, Embassies provides an ecosystem that resists pressure to expand the CEI, since developers can achieve arbitrary richness inside their picoprocesses.

In contrast, in today's web, many corporations still run Internet Explorer 6 for the sake of a single business-critical app. This compromise endangers all other apps on the client and the client system itself. In Embassies, the business app uses the Internet Explorer 6 DPI, which is no more (or less) dangerous to the client or her apps than a website that uses an old server-side library.

In addition to ecosystem-wide improvements, Embassies's design addresses specific web threats.

Cross-Site Request Forgery (CSRF). Today's CSRF attacks rely on the adversary's ability to trick the browser into sending out an app's cookies inappropriately (§4.2). The Embassies CEI never adds ambient authority [23] to an app's communications, so a banking app need never fear that the browser will blindly hand out its cookies.

Cross-Site Scripting (XSS). XSS flaws spring from poor library interfaces that fail to starkly distinguish data from the code that contains it; they are a client-side equivalent of server-side SQL-injection flaws. Embassies enables the vendor to migrate to rendering libraries that safely encapsulate tainted input, just as smart vendors use SQL libraries that safely encapsulate tainted input in `WHERE` clauses.

Clickjacking. Embassies resists clickjacking in the spatial domain by ensuring that each display region belongs to one viewport managed by only one app (§3.2). Vendors concerned about clickjacking in the temporal domain can implement client-side defenses, e.g., by ignoring inputs until 200ms after painting the display.

Side Channels. As in modern data centers, Embassies's pico-datacenter does not take steps to prevent side channels; i.e., one vendor may be able to infer another vendor's presence from the kernel's scheduling decisions or shared cache effects [2]. Current browsers face the same threats. Reducing the web's security problems to the existence of such side channels would be valuable progress.

Hosted Denial-of-Service. Embassies's minimality precludes it from reasoning about the Same-Origin Policy's content-based network restrictions; instead, Embassies addresses the underlying threats with CSN (§3.1.4). The consequence is developer freedom in network communication, but malefactors may abuse it to mount a denial-of-service (DoS) attack or a spam campaign. Today's web already allows such botnet-like attacks [34]; for example, to DoS a web server, the malefactor need only include a file (e.g., an image or JavaScript) in a popular website. Nonetheless, Embassies further enables such attacks. One mitigation would be for the client kernel to include a basic pushback mechanism [3] to allow remote hosts to squelch outbound traffic to the victim.

8 Discussion

Indexing and Mashups. Because the current web's CEI is so high-level, a vendor can easily create an app that interacts with other apps without their deliberate participation. A prominent example is web indexing, which works because the "internals" of most web content is in HTML. While Embassies permits vendors to use proprietary or obfuscated software, such behavior already occurs (e.g., Gmail's JavaScript code); baking HTML into the CEI does not guarantee hackability. In Embassies, HTML isn't required, but as with any popularity distribution, most apps will use one of a few popular DPI frameworks, and hence will allow third-party inspection. Because indexing is now so valuable, all popular DPI stacks will likely export an explicit indexing interface.

The "mashup" captures a broader category of serendipitous innovative reuse, such as data streams displayed on a map. Again, mashups interpose on the unobscured client-server traffic of ancestor applications; since those apps are likely to use popular frameworks, the same possibilities will be open. Ecosystem diversity is not enough to foil opportunistic extension; intentional obfuscation is required, a hurdle no less present in HTML than in Embassies.

Ad Blockers. Today, users can install browser extensions that interpose on apps. In Embassies, cooperating vendors could speak a bilateral protocol to a repository of extensions, but some extensions, like ad blockers, represent an adversarial relationship between user and vendor. Every user wants it, but no vendor does.

Since our CEI gives full control of an app to its the vendor, it confounds users who want to alter it in an unintended fashion. This tradeoff is deep. The client system cannot distinguish between an enhancement and a Trojan. Allowing extensions requires asking users to make that distinction, a responsibility few users can exercise correctly. We consider it worth giving up the ad blocker in exchange for a web where clicking links is always safe. Although this philosophy is new for the web, proprietary platforms such as the iPhone and Windows Phone deny unilateral app modifications.

Accessibility. Responsibility to provide accessibility falls to the vendor of each app, just as all aspects of app behavior do. However, we expect many vendors to write their applications against a higher-level DPI. Any mature DPI already incorporates accessibility features; thus any app built on such a DPI will be accessible.

Cross-Architecture Compatibility. Since our CEI specifies native-code execution, it does not solve the architecture portability problem in the CEI. We argue that architecture portability is a problem that can—and should—be solved in the vendor's software stack. One solution is to use a managed language (Java or .NET) or a portable representation (LLVM [37]) as a DPI.

DPIs based on unmanaged languages such as C or C++ can emit binaries for multiple architectures, as Linux distributions routinely do; this requires access to library source code (or recompiled libraries) as well. App vendors then face only the minor burden of hosting multiple binaries, a task easily automated, and less burdensome than dealing with today's browser incompatibilities.

On the rare occasion when a hardware company deploys a new Instruction Set Architecture (ISA), that ISA defines a new instance of the CEI. Until app vendors produce native binaries for the new ISA, the ISA company can implement, in their client kernel, an emulator for a popular ISA, as Apple did when it migrated its product line from 68K to PPC and again from PPC to x86.

GPUs. Today's web exploits the GPU by baking in further complexity, e.g., OpenGL or DirectX. Embassies' long-term solution is to treat the GPU as a CPU [7, 13, 21]. In the medium term, most deployed GPUs use segmented memory architectures adequate to isolate shader programs at GPU-load time without the client kernel understanding shader semantics. At present, even the CPU alone is pretty satisfying: Marble's CPU-rendered spinning globe (Fig. 5) is impressive.

Peripherals. Classic browsers expose printers and GPS. Does extending Embassies to include local devices erode the idea of the pico-datacenter? We think not.

Consider printing: Today, users can send photos from the Flickr app to the Snapfish app; Snapfish is a web service that includes a (remote) printer. Google Cloud Print extends the same semantics to a nearby printer. Indeed, many standalone printers already have IP interfaces. We can treat printers not as PC peripherals, but as applications that have a physical presence.

The same principle applies to other peripherals. A GPS with an IP interface need not be a PC peripheral; it may as well be an app like any other, one that gives the user control over which vendors see it. Of course, no IP hardware is required; the GPS can use a picoprocess on the client to host its IP stack.

Local storage is even simpler. Section 3.1.3 describes how apps employ a local untrusted storage service to securely store MACs and cookies. We have only implemented a RAM-based untrusted local store and a cloud-storage VFS module so far, but a disk could easily be exposed: Just a single vendor can manage the printer, Seagate might own the disk and offer untrusted, low-reliability storage, perhaps without even a UI.

Of course, we have described Embassies as a browser replacement, implying an underlying host OS; how does it interact with the host file system? Ultimately, we envision rich Embassies apps as a viable alternative to desktop OS apps. In the meantime, we envision exposing the host file system as another service, just as Google Cloud Print exposes the host printer as a service.

Deployment. Deploying a new web architecture is hard. However, Embassies apps can facilitate incremental deployment by providing a fallback for "legacy" HTTP links. With reference to Section 4.1, if caller.net's web runtime cannot resolve the name `target.org` using the PKI, it obtains and launches a web runtime which target.org might specify in a `browser.txt` file, or the caller app may supply a default.

This web runtime fetches and renders target.org's content via standard HTTP and HTML. However, the web runtime does not have a certificate chain for the label "target.org". Instead, the web runtime passes its own label (e.g., "mozilla.org") to `verify_label`. Thus, client kernel strongly authenticates the web runtime, which then attests, e.g., via its own intra-window decoration, that it is rendering content from target.org.

9 Conclusion

We propose to radically refactor the web interface to turn the client into a pico-datacenter in which app vendors run rich applications that are strongly isolated from each other. We described and implemented Embassies, a concrete, minimal CEI to support this vision, and we rebuilt existing browser-based app interactions atop the CEI. Our implementation and evaluation indicate that the CEI offers a significantly reduced TCB, yet supports a diverse set of DPIs. App and protocol performance is comparable to the existing web; app start time and memory usage is still higher than we would like, but there are clear paths towards improving them. Once native DPIs are available, and conventional apps can run in a web-like deployment, the Embassies architecture may become a compelling model for desktops or mobile platforms.

Acknowledgements

The authors gratefully thank Arun Seehra, Srinath Setty, and Xi Xiong, the interns who contributed to Embassies; Jay Lorch, James Mickens, and Mike Walfish for extensive discussions and feedback, with special thanks to James for suggesting the term "pico-datacenter"; Adrian Bonar for discussions and development work; and Alex Moshchuk, Will Scott, the anonymous reviewers, and our shepherd, Michael Piatek, for their helpful comments on the paper.

References

[1] Embassies source code. http://research.microsoft.com/embassies/, Feb. 2013.

[2] ABBOTT, R. P., CHIN, J. S., DONNELLEY, J. E., KONIGSFORD, W. L., TOKUBO, S., AND WEBB, D. A. Security analysis and enhancements of computer operating systems. Tech. rep., Institute for Computer Sciences and Technology, National Bureau of Standards, US Department of Commerce, Apr. 1976.

[3] ANDERSEN, D. G., BALAKRISHNAN, H., FEAMSTER, N., KOPONEN, T., MOON, D., AND SHENKER, S. Accountable Internet Protocol (AIP). In *Proceedings of ACM SIGCOMM* (2008).

[4] BARON, L. D. Preventing attacks on a user's history through CSS :visited selectors. http://dbaron.org/mozilla/visited-privacy, 2010.

[5] BARTH, A. HTTP state management mechanism. RFC 6265 (Proposed Standard), Apr. 2011.

[6] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Robust defenses for cross-site request forgery. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2008).

[7] BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. Brook for GPUs: Stream computing on graphics hardware. In *SIGGRAPH* (2004).

[8] COLLBERG, C., HARTMAN, J. H., BABU, S., AND UDUPA, S. K. SLINKY: Static linking reloaded. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (Apr. 2005).

[9] COX, R. S., GRIBBLE, S. D., LEVY, H. M., AND HANSEN, J. G. A safety-oriented platform for Web applications. In *IEEE Symp. on Security & Privacy* (2006).

[10] DHAMIJA, R., AND TYGAR, J. D. The battle against phishing: Dynamic security skins. In *ACM Symposium on Usable Security and Privacy (SOUPS '05)* (July 2005).

[11] DOUCEUR, J. R., HOWELL, J., PARNO, B., WALFISH, M., AND XIONG, X. The web interface should be radically refactored. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)* (2011).

[12] ESFAHBOD, B. *Preload: An adaptive prefetching daemon*. PhD thesis, 2006.

[13] FATAHALIAN, K., AND HOUSTON, M. A closer look at GPUs. *Communications of the ACM 51*, 10 (Oct. 2008).

[14] FESKE, N. Introducing Genode. Talk at the Free and Open Source Software Developers' European Meeting. Slide available at http://genode.org, Feb. 2012.

[15] FESKE, N., AND HELMUTH, C. A Nitpicker's guide to a minimal-complexity secure GUI. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2005).

[16] FESKE, N., AND HELMUTH, C. A nitpicker's guide to a minimal-complexity secure GUI. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2005).

[17] FESKE, N., AND HELMUTH, C. Design of the Bastei OS architecture. Tech. Rep. TUD-FI06-07, TU Dresden, Dec. 2006.

[18] GOOGLE. Google compute engine. http://cloud.google.com/compute/, 2012.

[19] GOSLING, J., JOY, B., AND STEELE, G. *Java™ Language Specification*. Addison-Wesley, 1996.

[20] GRIER, C., TANG, S., AND KING, S. T. Secure web browsing with the OP web browser. In *Proceedings of the IEEE Symposium on Security and Privacy* (2008).

[21] GUMMARAJU, J., MORICHETTI, L., HOUSTON, M., SANDER, B., GASTER, B. R., AND ZHENG, B. Twin peaks: A software platform for heterogeneous computing on general-purpose and graphics processors. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques (PACT)* (2010).

[22] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., SNOEREN, A. C., VARGHESE, G., VOELKER, G. M., AND VAHDAT, A. Difference engine: Harnessing memory redundancy in virtual machines. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Dec. 2008).

[23] HARDY, N. The Confused Deputy (or why capabilities might have been invented). *Operating Systems Review 22*, 4 (1988).

[24] HÄRTIG, H., HOHMUTH, M., LIEDTKE, J., WOLTER, J., AND SCHÖNBERG, S. The performance of μ-kernel-based systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (1997).

[25] HOWELL, J., DOUCEUR, J. R., ELSON, J., AND LORCH, J. R. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2008).

[26] HOWELL, J., JACKSON, C., WANG, H. J., AND FAN, X. MashupOS: Operating system abstractions for client mashups. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)* (May 2007).

[27] JACKSON, C., AND BARTH, A. Beware of finer-grained origins. In *Proceedings of the IEEE Web 2.0 Security and Privacy Workshop (W2SP)* (2008).

[28] JANG, D., VENKATARAMAN, A., SAWKA, G. M., AND SHACHAM, H. Analyzing the crossdomain policies of Flash applications. In *Proceedings of the IEEE Web 2.0 Security and Privacy Workshop (W2SP)* (2011).

[29] JOHNS, M. *Code Injection Vulnerabilities in Web Applications Exemplified at Cross-Site Scripting*. PhD thesis, University of Passau, 2009.

[30] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., NO, H. M. B., HUNT, R., MAZIÈRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. Application performance and flexibility on Exokernel systems. In *SOSP* (1997).

[31] KELLEY, P., CONSOLVO, S., CRANOR, L., JUNG, J., SADEH, N., AND WETHERALL, D. An conundrum of permissions: Installing applications on an android smartphone. In *Workshop on Usable Security* (2012).

[32] Kernel-based virtual machine. http://www.linux-kvm.org. Accessed May, 2012.

[33] KROVETZ, T., AND DAI, W. VMAC: Message authentication code using universal hashing. Internet Draft: http://http://fastcrypto.org/vmac/draft-krovetz-vmac-01.txt, Apr. 2007.

[34] LAM, V. T., ANTONATOS, S., AKRITIDIS, P., AND ANAGNOSTAKIS, K. G. Puppetnets: Misusing web browsers as a distributed attack infrastructure. In *Proceedings of the ACM Conference on Computer and Communications Security* (2006).

[35] LAMPSON, B., ABADI, M., BURROWS, M., AND WOBBER, E. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems 10*, 4 (Nov. 1992), 265–310.

[36] LANGLEY, A. Chromium's seccomp sandbox. Blog post, 2009. http://www.imperialviolet.org/2009/08/26/seccomp.html.

[37] LATTNER, C. LLVM: An infrastructure for multi-stage optimization. Master's thesis, UIUC, 2002.

[38] LEVIN, D., DOUCEUR, J. R., LORCH, J. R., AND MOSCIBRODA, T. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2009).

[39] MAHAJAN, R., PADHYE, J., RAGHAVENDRA, R., AND ZILL, B. Eat all you can in an all-you-can-eat buffet: A case for aggressive resource usage. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)* (2008).

[40] MERKLE, R. C. A certified digital signature. In *Proceedings of CRYPTO* (1989), pp. 218–238.

[41] MICKENS, J., AND DHAWAN, M. Atlantis: Robust, extensible execution environments for Web applications. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (2011).

[42] MICROSOFT MSDN. How to set network capabilities (Windows). http://msdn.microsoft.com/en-us/library/windows/apps/hh770532.aspx, Dec. 2012.

[43] MICROSOFT MSDN. Virtual machines. http://msdn.microsoft.com/en-us/library/windowsazure/jj156003.aspx, June 2012.

[44] MOSHCHUK, A., AND WANG, H. J. Resource management for web applications in ServiceOS. Tech. Rep. MSR-TR-2010-56, Microsoft Research, May 2010.

[45] MOSHCHUK, A., WANG, H. J., AND LIU, Y. Content-based isolation: Rethinking isolation policy in modern client systems. Tech. Rep. MSR-TR-2012-82, Microsoft Research, Aug. 2012.

[46] NILS. Building Android sandcastles in Android's sandbox. Black Hat, https://media.blackhat.com/bh-ad-10/Nils/Black-Hat-AD-2010-android-sandcastle-wp.pdf, Oct. 2010.

[47] O'NEILL, S. 'Consumerization of IT' taking its toll on IT managers. *CIO* (Sept. 2011).

[48] PARNO, B., LORCH, J. R., DOUCEUR, J. R., MICKENS, J., AND MCCUNE, J. M. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2011).

[49] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the Library OS from the Top Down. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[50] ROELOFS, G. libpng. Software distribution. http://www.libpng.org/pub/png/libpng.html.

[51] ROESNER, F., KOHNO, T., MOSHCHUK, A., PARNO, B., WANG, H. J., AND COWAN, C. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2012).

[52] SCHECHTER, S. E., DHAMIJA, R., OZMENT, A., AND FISCHER, I. The emperor's new security indicators. In *Proceedings of the IEEE Symposium on Security and Privacy* (2007), pp. 51–65.

[53] SHAPIRO, J. S., VANDERBURGH, J., NORTHUP, E., AND CHIZMADIA, D. Design of the EROS trusted window system. In *Proceedings of the USENIX Security Symposium* (2004).

[54] SIRER, E. G., DE BRUIJN, W., REYNOLDS, P., SHIEH, A., WALSH, K., WILLIAMS, D., AND SCHNEIDER, F. B. Logical attestation: An authorization architecture for trustworthy computing. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (2011).

[55] TANG, S., MAI, H., AND KING, S. T. Trust and Protection in the Illinois Browser Operating System. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2010).

[56] TWOTOASTS. Midori. http://twotoasts.de/index.php/midori/, 2012.

[57] WANG, H. J., FAN, X., JACKSON, C., AND HOWELL, J. Protection and communication abstractions for web browsers in MashupOS. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2007).

[58] WANG, H. J., GRIER, C., MOSHCHUK, A., KING, S. T., CHOUDHURY, P., AND VENTER, H. The multi-principal OS construction of the Gazelle web browser. In *USENIX Security Symposium* (2009).

[59] WANG, H. J., MOSHCHUK, A., AND BUSH, A. Convergence of desktop and web applications on a multi-service OS. In *USENIX HotSec Workshop* (2009).

[60] WANG, R., CHEN, S., AND WANG, X. Signing me onto your accounts through Facebook and Google: a traffic-guided security study of commercially deployed single-sign-on web services. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2012).

[61] WEBKIT. SunSpider JavaScript Benchmark. Version 0.9.1 at http://www.webkit.org/perf/sunspider/sunspider.html, 2012.

[62] The WebKit open source project. http://www.webkit.org/, 2012.

[63] WEINBERG, Z., CHEN, E. Y., JAYARAMAN, P. R., AND JACKSON, C. I still know what you visited last summer: User interaction and side-channel attacks on browsing history. In *Proceedings of the IEEE Symposium on Security and Privacy* (2011).

[64] WHEELER, D. A. SLOCCount. Software distribution. http://www.dwheeler.com/sloccount/.

[65] YE, E., AND SMITH, S. Trusted paths for browsers. In *Proceedings of the 11th USENIX Security Symposium* (Aug. 2002).

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

[67] ZALEWSKI, M. Browser security handbook: Same-origin policy. Online handbook. http://code.google.com/p/browsersec/wiki/Part2.